UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

ALGORITMOS PARA OPTIMIZAR CODIGO GENERADO POR UN COMPILADOR

TESIS PROFESIONAL

PARA OBTENER EL TITULO DE:

M A T E M A T I C O

P R E S E N T A

CARLOS RIVERA ORTEGA

DIRECTOR DE TESIS:

M. EN C. AMPARO LOPEZ GAONA

MEXICO, D. F. 1994

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

DEDICATORIA Y AGRDECIMIENTO.

Al cielo y al mar,

a mis padres y hermanos,

a la tierra y las estrellas,

a los maestros y alos amigos,

a la música y los colores,

a todo aquél que lea estas palabras

y que no la haya conocido;

a todos aquellos que son

mis padres, hermanos, maestros y amigos,

las tierras, los mares, los cielos y las estrellas

y que tal vez nos los haya conocido

aunque tenga su esencia (como música y color)

en el nombre de algunos de ustedes:

como mi papá Carlos Rivera Gonzalez (que es mi amigo, maestro y hermano);

mi mamá Altagracia Ortega Muñiz (que es mi maestra, tierra y cielo);

mi hermana Reyna Verónica Rivera Ortega (que es mi becerro de oro y mi cuataneta);

mis cuates: Mónica Leñero Padierna y Cia. S.A de C.V. & José Galavíz (estos dos últimos la Cia. mas querible); Mario Arturo Pérez Rangel y anexas ("Man-Arturo", "SuperArturo"); Norma Leticia González Aldama & Jesús Gutierrez García (estos dos otra S.A. de C.V.) y un largulsimo etc.;

mis maestros: Amparo López Gaona & Salvador López, Ana Luisa Solís (mis sinodales); mi maestra Conchita (mi primera maestra), Ma. de la Paz Alvarez, y un largísimo etc.

y a aquella quien solo yo soy capaz de soñar y guardar en mí porque es la substancia de todas las cosas que se esperan y el argumento de las que no aparecen en nuestra mente;

y a un larguísimo etc. entre parientes, amigos, maestros, y un conjunto inmenso de admiradoras.

A todos ustedes por las gracias que recibo con su existencia, por la confianza que tuvieron en que "*los algoritmos para optimizar código generado por un compilador*" tal vez sea útil (al menos para esta tésis).

INDICE GENERAL.

Indice general.

ALGORITMOS PARA OPTIMIZAR CODIGO GENERADO POR UN COMPILADOR.

0.- INTRODUCCIÓN

0.1.- PLANTEAMIENTO DEL PROBLEMA.

El cómputo es una de las ciencias con un gran desarrollo; cada tema relacionado con computación presenta mucha información (bibliografía, trabajos, etc.), la compilación y los compiladores no son la excepción; existe un gran número de trabajos acerca de compiladores, pero no todos desarrollan con profundidad el tema de la optimización del código generado por un compilador (no son muchos los trabajos dedicados a dicho tema). La razón por la que se hizo este trabajo no es nada más para aumentar la cantidad de información disponible acerca de un tema no muy conocido de los compiladores (la optimización del código que generan), sino para presentar herramientas útiles para la construcción de programas que optimizen código. Este texto es útil también para comprender el funcionamiento de un compilador ya que en el desarrollo del tema se explica dicho aspecto.

La computadora es una herramienta para la realización de procesos matemáticos; de igual manera que las máquinas mecánicas amplían las facultades físicas del humano, la computadora aumenta las capacidades mentales al realizar algunos de estos procesos muy rápido. Por eso es importante todo trabajo relacionado con estas máquinas.

Todo estudio acerca de las computadoras se relaciona con temas tales como la física, electrónica, matemáticas, ciencias sociales, literatura, etc. Esto se debe a que las computadoras cada vez son más utilizadas en las principales actividades humanas y porque el quehacer del hombre tiende a centrarse en el trabajo intelectual.

Por el avance del trabajo mental es que aparecieron las computadoras. Estos aparatos surgen de la necesidad de resolver ciertos problemas de matemáticas, debido a que la construcción y funcionamiento de estas máquinas tiene mucho que ver con esta disciplina. Ejemplo de esto son las primeras máquinas analíticas, calculadoras de tablas trigonométricas y sumadoras (antecedentes de las computadoras) que fueron construidas por matemáticos (Pascal, Babbage).

Muchos estudios o mejoramientos de las computadoras a lo largo de su historia surgen de la descripción, funcionamiento y aplicaciones de estas máquinas. En este escrito se estudiará el funcionamiento de las computadoras desde el punto de vista de un diseñador o constructor

de máquinas, pero no como un mecánico o ingeniero sino como los matemáticos que diseñaron las primeras computadoras, es decir, desde el punto de vista teórico.

El objetivo de este trabajo es mostrar cómo se puede mejorar el funcionamiento de una computadora a partir de los elementos que permiten la comunicación de estas máquinas con los hombres. Esto es, como en toda computadora existen **mecanismos** para permitir que las personas puedan utilizar estas máquinas, se describirá uno de estos **mecanismos** y se presentarán los métodos que permiten obtener los mejores resultados.

Para describir más precisamente el objetivo de este trabajo se comienza definiendo los conceptos necesarios.

0.2.- CONCEPTOS BÁSICOS.

COMPUTADORA.

Una computadora es una máquina que, esencialmente, resuelve problemas matemáticos, simula procesos y ejecuta tareas definidas por medio de instrucciones (programas). El conjunto de instrucciones que puede ejecutar una computadora se llama **lenguaje de máquina**. Para el manejo de estos aparatos es necesario conocer las instrucciones de su lenguaje de máquina y la manera como pueden funcionar; al manejo de una computadora como consecuencia de la comprensión de su funcionamiento se llama comunicación hombre-máquina.

Una característica que ha hecho de la computadora una de las máquinas más útiles es que el proceso de comunicación es relativamente sencillo, esto se debe a que en las computadoras existen "mecanismos" para facilitar la comunicación; dichos "mecanismos" simulan una máquina (llamada **máquina virtual**) definida con "instrucciones sencillas" para los usuarios de la computadora, y parte de la simulación incluye un transformador de "las instrucciones sencillas" a las instrucciones definidas en la computadora real. Estos "mecanismos" surgen debido a que que la mayoría de los lenguajes de máquina son muy elementales y su uso es "difícil y tedioso".

TRADUCCIÓN E INTERPRETACIÓN.

La naturaleza de los lenguajes de máquina está dada por la tecnología existente, los tipos de arquitectura de las computadoras y, en menor grado, por la necesidad de construir computadoras

"baratas y fáciles de usar". Los "mecanismos" implantados en las máquinas para facilitar su uso y comprensión forman parte de la comunicación hombre-máquina. En dicho proceso existe un programa encargado de hacer la transformación del lenguaje de la computadora simulada al lenguaje de máquina. Hay dos formas de hacer dicha "transformación": la interpretación y la traducción

La interpretación en la comunicación hombre-máquina consiste en tomar cada instrucción del lenguaje fuente[1] y ejecutarla en instrucciones equivalentes de la máquina.

La traducción consiste en convertir todo conjunto de instrucciones de la computadora simulada en un programa equivalente en lenguaje de máquina. Los programas que hacen traducciones se llaman ensambladores o compiladores. La interpretación y traducción se parecen mucho pero la diferencia entre ellos es que un traductor produce un programa que se puede usar después independientemente del traductor, mientras que usando un intérprete cada vez que se quiera la ejecución del programa se tiene que usar al intérprete. En este sentido los compiladores son mejores que los intérpretes y por esta razón es que en este trabajo se hablará de ellos.

HARDWARE Y SOFTWARE

Una computadora es una máquina que ejecuta programas y para hacerlo necesita componentes físicos y lógicos. El *hardware* es el conjunto de partes tangibles de la máquina (componentes físicos). El *software* es el conjunto de todos los programas que se ejecutarán en la máquina (componentes lógicos). Existen muchos tipos de *software*: el encargado del funcionamiento de la máquina, el que se dedica a la comunicación hombre-máquina, el que los usuarios desarrollan, etc.

La expresión del *software* en la máquina son los programas, y en esta parte es donde se encuentra el objeto de estudio de este trabajo porque el compilador es un programa y es parte del *software* de las computadoras.

PROGRAMA.

Un **programa** es un conjunto de instrucciones escritas en un lenguaje que describen cómo hacer una tarea de acuerdo a un cierto algoritmo. Máquinas y seres vivos ejecutan programas. Cada ser que puede ejecutar un programa tiene un lenguaje para entender las instrucciones: así,

[1] Se llama así al lenguaje que se transformará y los programas escritos en dicho lenguaje se llaman **programas fuentes**

se tienen los idiomas para las personas, el código genético para los seres vivos y los lenguajes de programación para las computadoras.

COMPILADOR.

Para que una computadora realice las instrucciones deseadas se necesita conocer el lenguaje que la máquina puede "entender". Esto es porque cada computadora solamente puede realizar sus funciones si se le ordena hacerlo de acuerdo a su lenguaje de máquina.

Se pueden hacer programas para máquinas escribiéndolos en sus idiomas, pero esto es muy difícil porque estos lenguajes son complicados, debido a que todo lenguaje máquina tiene como característica la de manejar instrucciones básicas relacionadas directamente con la arquitectura de la máquina y debido a lo elemental que son su uso resulta difícil y muy tedioso; otra desventaja es que si existen muchos tipos de máquinas se tendrían que aprender muchos lenguajes diferentes haciendo muy complicado el uso de computadoras. Para evitar estos problemas, existe como *software* el programa **compilador** que es traductor de un lenguaje "fácil" de entender por los programadores al lenguaje máquina.

Existen muchos tipos de compiladores y todas las computadoras modernas tienen estos programas, cuya función es la misma, traducir. La existencia de muchos compiladores se debe al gran número de lenguajes de programación que aparecen continuamente.

La traducción a un lenguaje de máquina hecha por el compilador dará como resultado instrucciones o código en lenguaje que "entenderá" y ejecutará la máquina. Estas nuevas instrucciones forman el **programa objeto** que es la traducción del **programa fuente**. La ejecución del programa objeto (o programa traducido al lenguaje máquina) en la computadora será la mejor según los criterios que se apliquen, por ejemplo si el código generado por un compilador es equivalente a otro más pequeño entonces dicho compilador no será el mejor si se quiere ahorrar espacio, pero si el código grande se ejecuta más rápido en la computadora que el código chico entonces el compilador que lo generó es el mejor bajo el criterio de la velocidad de ejecución.

En este trabajo se describen métodos para hacer que las traducciones hechas por los compiladores sean las mejores en el sentido de velocidad y espacio.

CAPITULO UNO

1.- TEORÍA DE COMPILADORES

1.0.- INTRODUCCIÓN

Antes de construir una computadora se debe resolver el problema de definir el tipo de instrucciones y el lenguaje que usará. Estos (el lenguaje e instrucciones) serán de tal forma que los usuarios de la máquina los comprendan con facilidad y que sean "baratos y fáciles" de implantar.

El problema de la definición de un lenguaje apropiado para la máquina y sus usuarios se resuelve creando dos lenguajes: uno "adecuado" para la máquina y otro "apropiado" para los usuarios y construyendo un convertidor del lenguaje del usuario al lenguaje de máquina.

MÁQUINAS VIRTUALES

Teniendo un convertidor de instrucciones los usuarios manejan a la computadora como si el lenguaje de la máquina fuera el lenguaje del usuario; al hacer esta suposición se define una máquina hipotética o virtual[2] porque aparentemente la computadora ejecuta programas escritos en lenguaje del usuario aunque en realidad sólo reconozca el lenguaje de máquina.

Toda computadora real define un lenguaje (el lenguaje máquina) y también toda computadora virtual que inventemos establece un lenguaje. De la misma manera, un lenguaje define una máquina: la que puede ejecutar los programas escritos en ese lenguaje, ya sea real o virtual. [3] [Hopcroft et. al. 1979]

MÁQUINAS MULTINIVEL

Para hacer el programa que convierta programas escritos en **lenguaje usuario** (L_u) a **lenguaje de máquina** (L_m), L_u y L_m no deben ser muy diferentes para que el programa convertidor no sea muy difícil de construir. Este nuevo problema se resuelve creando un lenguaje intermedio L_1 que es parecido a L_u y a L_m, y con el nuevo lenguaje intermedio también se crea su correspondiente convertidor de instrucciones. Si el nuevo transformador de

[2] Una máquina virtual es una computadora hipotética que ejecuta programas escritos en un lenguaje también hipotético

[3] Para todo lenguaje existe una máquina de Turing que lo representa.

lenguajes sigue siendo complicado entonces se repite el proceso anterior, y se crea un lenguaje L_2 con su convertidor de L_1 a L_2. Este método se repite tantas veces como se requiera y consiste en crear un lenguaje entre L_u y el último L_i. En este proceso cada L_i creado es parecido al anterior L_{i-1} y al lenguaje usuario L_u por lo que los transformadores convierten instrucciones de Li a L_{i-1} (para toda i).

Se crean tantos L_i's como sean necesarios con sus respectivos programas convertidores entre cada uno y así se resuelve el problema porque al final de este proceso se tiene la conversión de instrucciones que "entiende" el hombre a su equivalente por las que puede realizar la computadora.

El número de lenguajes intermedios creados determina la complejidad del lenguaje usuario y del tipo de comunicación.

Como un lenguaje define una máquina, en este caso por cada L_i o nivel se establece una máquina virtual. Y este proceso se sigue en la construcción de computadoras: se parte del lenguaje de máquina (nivel 1) y se crean tantos niveles como se necesiten hasta llegar a un lenguaje que entiendan los usuarios (el nivel más alto).

Los niveles más bajos son los cercanos al lenguaje de máquina y los niveles más altos son los cercanos al lenguaje de usuario.

PROGRAMAS QUE CONVIERTEN INSTRUCCIONES

Cada programa convertidor transforma instrucciones de L_j a L_i donde $i < j$. Y un convertidor del nivel j al nivel i puede estar escrito en L_i, L_j o cualquier otro lenguaje L_k. Generalmente el traductor está escrito en un lenguaje L_k donde $i < k < j$.

Como ya se dijo, los programas que convierten las instrucciones de un lenguaje L_b a otro L_a ($a < b$) siguen dos métodos principalmente: la interpretación y la traducción.

La interpretación consiste en que para toda instrucción en L_b se ejecuta una o más instrucciónes equivalentes escritas en L_a. Los programas intérpretes de L_b a L_a se escriben en L_a.

La diferencia entre un traductor y un intérprete es que al traducir de L_b a L_a se crean programas en L_a para ser ejecutados después, mientras que al interpretar se ejecuta directamente cada instrucción de L_b en L_a.

Generalmente los intérpretes se encuentran en los niveles más bajos de lenguaje mientras que los traductores se encuentran en los niveles más altos, aunque existen intérpretes de lenguajes de alto nivel.

Un compilador es un traductor y estos programas son muy empleados debido a la gran variedad de lenguajes de alto nivel especializados que existen.

1.1.- PARTES DE UN COMPILADOR

ESTRUCTURA Y FUNCIONAMIENTO DE UN COMPILADOR

La función de traducción de todo compilador se divide en dos partes : análisis y síntesis.

La parte de **análisis** de un compilador se encarga de revisar que los programas a traducir estén correctamente escritos de acuerdo a la definición del lenguaje fuente.

Para revisar si un programa está bien escrito se debe verificar que las reglas del lenguaje se cumplan en cada parte del programa. Las normas de un lenguaje se dividen en leyes particulares y generales. Las reglas particulares indican cómo se debe construir cada expresión sencilla (en este caso cada instrucción); y las leyes generales indican cómo debe definirse el conjunto de expresiones sencillas. Entonces para que un compilador verifique el cumplimiento de las leyes de un lenguaje debe comparar el programa a analizar con las leyes particulares y luego con las leyes generales. Por lo tanto la parte del análisis de un compilador se divide en dos: revisión de leyes particulares y revisión de leyes generales del lenguaje.

La parte de **síntesis** trata de la traducción y mejora de las instrucciones obtenidas en la etapa anterior, y todo esto se hace de acuerdo a la información obtenida del análisis. En la síntesis se requiere de técnicas especializadas, la optimización de código generado que es el tema de este trabajo es una de estas técnicas.

1.1.1.- ANÁLISIS

En esta parte el compilador verifica el cumplimiento de las reglas del lenguaje en los programas a traducir. Para especificar de qué manera hace esto el traductor, se deben definir las reglas locales y globales del lenguaje.

Las reglas locales de un lenguaje se conocen como el **léxico.** El **léxico** de un lenguaje es el conjunto de reglas que definen el tipo de símbolos y la manera en que pueden agruparse; esto es, indica cuáles son las palabras válidas del lenguaje.

Las reglas que definen las combinaciones válidas entre las palabras de un lenguaje se llaman **gramática**. [4] La gramática está formada por el léxico, la sintáxis y la semántica.

Las reglas globales de un lenguaje se dividen en la sintaxis y la semántica.

La **sintaxis** se refiere a las leyes del lenguaje que indican la manera en que se pueden agrupar las expresiones válidas según el léxico. Esto es, la sintaxis indica cuales conjuntos de palabras son correctos (como las oraciones).

La **semántica** establece cómo pueden agruparse las expresiones sintácticamente válidas, es decir, determina "que oraciones tienen significado válido".

Por lo tanto la parte de análisis en un compilador se compone del análisis de la gramática, esto es, **análisis léxico, sintáctico y semántico.**

Para poder construir un programa traductor se deben conocer las reglas o leyes que forman la gramática del lenguaje que se traducirá, y estas leyes deben ser claras y no ambiguas.

ANÁLISIS LÉXICO

En el análisis léxico los traductores revisan todos los símbolos del programa fuente (alfabeto) y comprueban que estén definidos y su agrupación sea correcta.

Un léxico esta formado por **componentes léxicos** que son los símbolos (dígitos, letras y palabras) válidos para el lenguaje. Para construir un programa que reconozca componentes léxicos es necesario poder expresar todas las combinaciones permitadas de todos los símbolos válidos.

La forma en que se especifican los componentes léxicos es por medio de **expresiones regulares**; este método describe la manera como se produce cada combinación válida a partir de un conjunto de símbolos definidos [5].

---

[4] Una gramática describe la estructura jerárquica de muchas construcciones de los lenguajes de programación.

[5] Las gramáticas regulares se forman de las expresiones regulares [Hopcroft, et. al. 1979]

Existen reglas para construir expresiones regulares. Una expresión regular se construye a partir de expresiones regulares simples.

Cada expresión regular γ representa un lenguaje llamado L_γ. Si se tiene un alfabeto Σ, entonces las reglas que definen expresiones regulares a partir del alfabeto son:

- La expresión vacía ϵ es expresión regular.

- Si a pertenece al alfabeto Σ, entonces a es una expresión regular.

- Si a y b son expresiones regulares entonces:

  a) $a|b$ es una expresión regular representada por $L_a \cup L_b$.

  b) ab es una expresión regular representada por $L_a L_b$.

  c) a^* es una expresión regular representada por L_a^*.

  (a^* significa la concatenación de a consigo misma (cero o más veces)).

La manera de construir programas que reconozcan expresiones regulares es usando **autómatas**. Un autómata es un modelo matemático; los tipos de autómatas que se usan en los reconocedores de léxicos son los autómatas finitos deterministicos [6].

ANÁLISIS SINTÁCTICO

El análisis sintáctico se encarga de revisar que el conjunto de componentes léxicos estén bien agrupados, esto es, que las combinaciones de las "palabras" sean válidas.

[6] Este modelo se forma de un conjunto de estados, un alfabeto y una función que mueve o transforma los pares de símbolos-estados: por cada símbolo reconocido, un estado se "transforma" a otro. Los autómatas finitos deterministicos son un caso particular de los autómatas finitos no deterministicos

Para construir en el compilador un analizador sintáctico las reglas que definen a la sintaxis deben definirse por medio de una gramática independiente del contexto.

Las gramáticas independientes del contexto se emplean en la especificación de la sintaxis para la construcción de un traductor debido a que describen de manera natural la estructura de muchos lenguajes de programación, además estas gramáticas permiten la fácil generación de código y la construcción de nuevas definiciones de la sintaxis.

Una gramática independiente del contexto tiene las siguientes partes:

- Una colección de símbolos terminales (símbolos básicos con que se forman las cadenas como los componentes léxicos que son los símbolos o palabras válidas en el lenguaje como palabras las reservadas y las no reservadas).

- Un conjunto de símbolos no terminales (todos los símbolos que no son terminales y que identifican a una producción, es decir, las variables que identifican al conjunto de cadenas)

- Reglas o producciones.

- Un símbolo inicial.

Las reglas se forman de dos partes: la izquierda y la derecha (en general cada lado se separa por una flecha), del lado izquierdo se encuentran símbolos no terminales; y del lado derecho existen símbolos terminales, no terminales o combinaciones de ellos. El símbolo inicial siempre es un no terminal. Las producciones especifican cómo combinar terminales y no terminales para formar cadenas o "frases" dentro del lenguaje.

EJEMPLO:

Sean las siguientes producciones de una gramática:

$expr \rightarrow expr\ op\ expr$

expr -> *id*

op -> +

En esta gramática los símbolos terminales son *id* y +

Los símbolos no terminales son *expr* y *op*; *expr* es además el símbolo inicial.

Una forma para reconocer el lenguaje formado por una gramática independiente del contexto es usando árboles sintácticos [7].

ANÁLISIS SEMÁNTICO

El análisis semántico verifica que el significado de las estructuras reconocidas por los otros análisis sea válido.

Para implantar el análisis semántico en el compilador se utiliza la construcción de los analizadores léxicos y sintácticos. A cada componente léxico se le asocia un conjunto de atributos o características (por ejemplo números) para que cuando dichos componentes léxicos se acomoden en las producciones definidas por el análisis sintáctico se obtenga una determinada combinación de atributos, y la semántica define si estas combinaciones son válidas (y el análisis semántico se encarga de verificarlas). A esta manera de analizar la semántica se le llama traducción dirigida por la sintaxis.

Para implementar lo anterior existen herramientas de *software*. Hay programas que reconocen expresiones regulares, [8] que construyen autómatas [9], etc.

[7] Se llama árbol a la gráfica formada por un nodo (o raíz) del que parten una o mas ramas (o aristas) hacia otros nodos (en un árbol no existen ciclos). En un árbol sintáctico las ramificacines las determinan las producciones.

[8] En UNIX existe el programa llamado *lex* que reconoce expresiones regulares y realiza el análisis léxico

[9] En UNIX el programa *yacc* construye autómatas para el análisis sintáctico

1.1.2.- SÍNTESIS

GENERACIÓN DE CÓDIGO Y OPTIMIZACIÓN

Un compilador se forma por los analizadores de léxico, de sintaxis y de semántica, pero con solo estos elementos todavía no se tiene un traductor de lenguajes. Una vez que el compilador ya verificó la validez de un programa el siguiente paso es hacer la traducción propiamente dicha. Este paso lo hacen muchos compiladores incluyendo las instrucciones de traducción en sus analizadores, esto es, algunos analizadores de sintaxis, de semántica o, inclusive, de léxico, pueden incluir rutinas de generación de código ya traducido.

Un traductor de programas puede estar formado nada más por los analizadores y el generador de código, pero las necesidades de tener *software* y *hardware* barato hacen que se agregue otra parte a los compiladores, la parte de optimización del código generado por el compilador. La optimización del código toma la traducción hecha por el compilador y la mejora.

Como el programa traducido por el compilador será la entrada para la máquina virtual de un nivel inferior, en esa computadora virtual el código tendrá un tamaño y un tiempo de ejecución. El objetivo de la optimización de código es generar programas que ocupen el menor espacio posible y cuyo tiempo de ejecución sea lo más rápido. Para lograr esto se utilizan algoritmos que resuelven el problema de tener rapidez máxima o tamaño mínimo (aunque la existencia de algoritmos que den máxima rapidez y menor tamaño en la generación de código es un problema indecidible).

CAPITULO DOS

2.- TRANSFORMACIONES Y ANÁLISIS DE CÓDIGO.

2.0.- INTRODUCCIÓN.

En la tarea de mejorar todo programa traducido es necesario hacer cambios o transformaciones de tal manera que no se modifique el funcionamiento del programa, y para lograr esto se requiere hacer un análisis del código.

En este capítulo se exponen los métodos empleados en el mejoramiento de tiempo de ejecución y tamaño de programas; estos procedimientos son utilizados por los algoritmos para la optimización de código.

ANÁLISIS DE CÓDIGO

Por analizar código se entiende el conjunto de pasos empleados para determinar la estructura y funcionamiento del programa, es decir, la descripción de métodos que indiquen las partes de un programa y su utilidad en la ejecución de las instrucciones. Conociendo estos datos se pueden aplicar transformaciones para mejorar el código.

TRANSFORMACIONES

Una transformación de código es un método, un conjunto de pasos o criterios que permiten obtener la equivalencia de un grupo de instrucciones a otro. Las transformaciones que se analizan son aquellas que, de alguna manera, mejoran la velocidad de ejecución de programas o reducen el tamaño del código.

Un compilador que optimiza utiliza transformaciones basadas en el análisis del código que genera.

2.1.- TRANSFORMACIONES.

2.1.1.- ELIMINACIONES.

Los cambios que se pueden hacer a todo programa sin modificar su funcionamiento en cuanto a los resultados que produce, son:

- Eliminación de subexpresiones comunes.
- Eliminación de instrucciones redundantes.
- Eliminación de código inactivo.
- Traslado de código.
- Transformación algebraica.
- Propagación de copias.

Estas transformaciones son válidas para todo tipo de código, es decir, no importa el lenguaje en que esté escrito el programa, siempre se podrán hacer estas transformaciones sin que se altere su funcionamiento (esto es, el programa seguirá dando los mismos resultados).

ELIMINACIÓN DE SUBEXPRESIONES COMUNES.

La eliminación de subexpresiones comunes consiste en sustituir, en cada instrucción, toda expresión ya calculada o definida.

Ejemplo 2.1 :

Aplicación de la transformación en un código pequeño

(1) $a := b+c$	(1*) $a := b+c$
(2) $g := i*j$	(2*) $g := i*j$
(3) $c := a+e$	(3*) $c := a+e$
(4) $f := b+c$	(4*) $f := b+c$
(5) $h := i*j$	(5*) $h := g$

En la instrucción (5) se asigna el mismo cálculo que en (2) por eso la transformación hace $h:=g$. Pero a la instrucción (4) no se le aplica el cambio de este criterio, aunque tenga la misma operación $b+c$ que en (1), porque la variable c no tiene el mismo valor debido a la instrucción (3) $c:=a+e$; es decir, las instrucciones (1) y (4) no contienen una expresión común aunque efectúen la misma operación con las mismas variables porque las variables involucradas no tienen el mismo valor.

ELIMINACIÓN DE INSTRUCCIONES REDUNDANTES.

Este cambio en el código elimina todas las repeticiones de una instrucción siempre y cuando tengan los mismos valores que el original.

Ejemplo 2.2 :

Sea el siguiente conjunto de instrucciones y su transformación:

(1) A : = B	(1*) A : = B
(2) C : = D	(2*) C : = D
(3) D : = 7	(3*) D : = 7

(4) A : = B (4*) C : = D

(5) C : = D

La instrucción *(4)* se eliminó porque es igual que *(1)*, pero *(5)* no se quita aunque sea igual a la instrucción *(2)* ya que el valor que se asigna a *C* en *(2)* no es el mismo en *(5)* porque *D* cambió su valor debido a *(3)*.

ELIMINACION DE CÓDIGO INACTIVO

Esta transformación elimina del programa las instrucciones usadas sólo una vez y que no están relacionadas con otras partes del código; por ejemplo, no se toma en cuenta para la eliminación a instrucciones que modifican el orden de ejecución del código e instrucciones de asignación de variables que son usadas en otras partes del código.

Ejemplo 2.3 :

Aplicar el criterio en el siguiente código:

(1) $u := 0$

(2) $y := 1 + u$

(3) $x := 17$

(4) ir a la instrucción (6)

(5) $y := v$

(6) si y <7 ir a la instrucción (2) [9]

[9] Este salto quiere decir que se repetirán las instrucciones (2), (3) y (4) mientras el valor de y sea menor a 7.

Eliminando código inactivo se obtiene:

(1*) $u := 0$

(2*) $y := 1 + u$

(3*) **si** $y < 7$ **ir a la instrucción (2)**

Se suprimen las instrucciones (4) y (5) debido a que nunca se utiliza la instrucción (5) como consecuencia de la instrucción de salto en (4). De este código también se podría eliminar la instrucción (3) porque la variable x tampoco está relacionada con la ejecución del resto del código, pero al hacer esto se corre el riesgo de quitar variables útiles para todo el programa (si el programa consta de más código que del ejemplo), pero si el programa fuera nada más el código de este ejemplo y no interesara el valor de x, entonces si se elimina (3).

2.1.2.- TRASLADOS.

Hay dos tipos de traslado de código: el intercambio de proposiciones adyacentes independientes y el traslado de invariantes en ciclos.

Dado un conjunto de instrucciones, en ciertos casos se puede cambiar el orden de ejecución sin que se altere el funcionamiento del programa. Cuando se tienen dos o más instrucciones que no tienen relación alguna y están juntas se pueden intercambiar. Este es el intercambio de proposiciones adyacentes independientes.

Ejemplo 2.4 :

Sea el siguiente código

(1) $a := b + c$

(2) $d := e + f$

Aplicando aquí esta transformación se obtiene

(1*) $d := e + f$

(2*) $a := b + c$

Se intercambió la instrucción (1) por (2).

El intercambiar dos proposiciones no reduce el tamaño del código y en cuanto a la velocidad sólo en algunos casos este tipo de transformaciones podría disminuir el tiempo de ejecución; si este intercambio logra que todas las instrucciones que utilizan siempre las mismas variables estén juntas y que las variables menos usadas queden aparte, entonces la ejecución del código será más veloz si la computadora tiene como característica mantener en memoria sólo las variables más usadas.

El traslado de invariantes en ciclos consiste en mover código de un conjunto de instrucciones que se repiten más de una vez en todo el programa. El código trasladado de estos bloques de instrucciones tiene como característica que no modifica el valor de sus variables con cada repetición del bloque.

Ejemplo 2.5 :

(1) $b := -1$

(2) $a := 1$

(3) $a := b + 1$

(4) $c := 17$

(5) $d := a$

(6) Si $d < 5$, entonces ir a la instrucción (3)

El bloque de instrucciones (3), (4), (5) y (6) se repite 5 veces. Con el traslado de código se obtiene:

(1 *) $b := -1$

(2 *) $a := 1$

(3 *) $a := b+1$

(4 *) $d := a$

(5 *) Si $d < 5$, entonces ir a la instrucción (3 *)

(6 *) $c := 17$

El cambio consistió en mover la instrucción **(4)** $c := 17$ fuera del bloque de código que se repite (la secuencia que va desde la asignación **(3)** $a := b+1$ hasta la instrucción **Si** $d<5$, **entonces ir a la instrucción (3)**).

Esta transformación se pudo hacer sin alterar los resultados obtenidos por este código porque el valor de c no se modifica con la ejecución del ciclo (es una invariante del ciclo) y porque el valor de c no interviene en las instrucciones (3) y (5) que sí son afectadas por cada repetición.

El trasladar invariantes de ciclos no reduce el tamaño del código pero si aumenta la velocidad de ejecución porque así se reduce el tamaño de los bloques de instrucciones que se repiten.

2.1.3.- ALGEBRAICAS.

Este tipo de transformaciones están basadas en la aplicación de las leyes del álgebra para todo código o instrucción formado por expresiones algebraicas.

En todo programa es común que aparezcan expresiones como sumas, restas, multiplicaciones o cualquier tipo usual de operación matemática de variables o números que siempre son expresiones algebraicas, y al cambiar estas expresiones por otras equivalentes se está haciendo una transformación algebraica. Debido al uso de métodos matemáticos estas modificaciones no cambian el resultado de los programas.

Las transformaciones algebraicas para el propósito de optimización de código se clasifican en uso de identidades, reducción de intensidad y cálculo previo de constantes.

IDENTIDADES ALGEBRAICAS

Para muchas operaciones algebraicas existen elementos que dejan invariantes las operaciones, por ejemplo en la suma y resta es el cero ($a+0=a$, $a-0=a$), en la multiplicación y división es el uno ($a*1=a$, $a/1=a$); estos números (el cero y el uno) se llaman **elementos identidad**. La transformación llamada identidad algebraica consiste en localizar todos los elementos identidad en cada operación y sustituir la operación por su equivalente. Se menciona este criterio no porque los programadores escriban identidades algebraicas sino porque en el proceso de compilación, se pueden producir expresiones algebraicas con identidades no simplificadas.

Ejemplo 2.6 :

Aplicar el uso de identidades algebraicas al siguiente código:

$$(1)\ A := A+X+0$$

$$(2)\ C := (A/1)+2$$

$(3)\ C := C - 0 + A$

Localizando todas las identidades algebraicas este código queda así:

$(1^*)\ A := A + X$

$(2^*)\ C := A + 2$

$(3^*)\ C := C + A$

La transformación consistió en eliminar al cero en (1) $(X + 0 = X)$, eliminar la división por uno de la instrucción (2) $(A/1 = A)$, y eliminar el cero de la instrucción (3) $(C - 0 = C,\ 0 + A = A)$. En un buen programa no deben aparecer identidades sin simplificar, pero cuando se optimiza código es útil tener en cuenta esta transformación.

REDUCCIÓN DE INTENSIDAD

Reciben el nombre de reducción de intensidad las transformaciones que utilizan la igualdad entre operaciones algebraicas para sustituir "operaciones complicadas" por su equivalente en "operaciones sencillas", y al referirse a la complejidad de operaciones se definen las operaciones matemáticas sencillas como la suma y resta, y como complicadas todas las demás.[11] El orden de complejidad es el siguiente: suma, resta, multiplicación, división, potenciación.

Ejemplo 2.7 :

Sea el siguiente código y aplíquese el criterio de reducción de intensidad para las operaciones algebraicas:

$(1)\ A := X^3$

[11] Esta definición es así debido a que como se usarán estas transformaciones para optimizar código que ejecutará una computadora, para estas máquinas las operaciones más sencillas son las sumas y restas. Y por esta razón recibe el nombre de reducción de intensidad, porque se reduce la intensidad de las operaciones.

$(2)\ B := X * 2$

$(3)\ C := X/5$

La operación X^3 quiere decir que X se eleva al cubo. Aplicando el criterio de escribir las expresiones en término de operaciones sencillas el código queda así:

$(1^*)\ A := X * X * X$

$(2^*)\ B := X + X$

$(3^*)\ C := X * 0.2$

La transformación consistió en lo siguiente: en (1) X^3 se sustituyó por $X * X * X$ (elevar al cubo es mas complicado que multiplicar); en (2) $X * 2$ se sustituyó por $X + X$; en (3) $X/5$ se sustituyó por $X * 0.2$.

El cambio de una exponenciación por multiplicaciones se hará cuando el exponente sea un natural (no una variable); se sustituirá una multiplicación por sumas cuando se multiplique un número por un natural, y se cambiará una división por una multiplicación cuando un número sea dividido por un real[12].

CÁLCULO PREVIO DE CONSTANTES

Esta transformación consiste en efectuar todas las operaciones no realizadas entre números, es decir, se sustituye todo cálculo por el resultado explícito siempre y cuando éste (el cálculo) no involucre variables.

[12] En algunos casos el proceso de transformar una división por multiplicaciones será equivalente a realizar la división (por ejemplo, para convertir $x/3.5$ a una multiplicación es necesario realizar la división de $1/3.5$), pero el código ya compilado será más rápido de ejecutar (para las computadoras es más "fácil" multiplicar que dividir) y ese es el objetivo de estas técnicas.

Ejemplo 2.8 :

Aplicar el cálculo previo de constantes al siguiente código:

$$(1)\ X := 2 * 3$$

$$(2)\ Y := X + 5 - 2$$

$$(3)\ Z := (9 + 8) * Y$$

Se obtiene:

$$(1^*)\ X := 6$$

$$(2^*)\ Y := X + 3$$

$$(3^*)\ Z := 17 * Y$$

La transformación consistió en efectuar $2 * 3$ en (1), $5 - 2$ en (2) y $9 + 8$ en (3).

PROPAGACIÓN DE COPIAS.

Por copia se entiende la asignación de una variable a otra *($A := B$)*, pues el asignar el valor de B a la variable A provoca la copia del valor de B en A (y con esto cuando se necesite dicho valor se tendrán dos variables para una misma entidad desperdiciándose espacio y velocidad en la ejecución del programa, porque en lugar de repetir un mismo valor en dos o más variables es preferible mantener un solo valor en una variable).

La transformación de código basada en la propagación de copias consiste en utilizar siempre la variable asignada B en lugar de A, en toda instrucción que utilice el valor de A, (esto siempre y cuando no se modifiquen nuevamente los valores de A y B).

Ejemplo 2.9 :

Supóngase que el programa es el siguiente fragmento de código:

$$(1)\ A := B$$

$$(2)\ C := D * E$$

$$(3)\ E := A$$

La transformación sobre propagación de copias deja al código así:

$$(1^*)\ A := B$$

$$(2^*)\ C := D * E$$

$$(3^*)\ E := B$$

El cambio consistió en sustituir en (3) A por B. Esta transformación siempre se completa con la eliminación de código inactivo, así el ejemplo completo quedaría así:

$$(1^{**})\ C := D * E$$

$$(2^{**})\ E := B$$

La nueva modificación consistió en eliminar a (1*) usando la eliminación de código inactivo.

2.1.4.- APLICACIONES.

Al aplicar las transformaciones mencionadas a cualquier código programable, no se modifica el funcionamiento [13]. Como el propósito es optimizar código producido por un compilador, los programas que se considerarán para aplicar transformaciones que puedan mejorarlos estarán escritos en **código de tres direcciones**[14] y este código permite el análisis de los programas para poder aplicar transformaciones que los mejoren, esto es, a partir del código de tres direcciones se construyen estructuras útiles para implantar transformaciones que optimicen el código.

[13] Estas modificaciones transforman programas en otros equivalentes.

[14] El código de tres direcciones es parecido a muchos lenguajes ensambladores por lo que su traducción hacia uno de ellos no es difícil.

2.2.- TÉCNICAS QUE PERMITEN EL ANÁLISIS DE CÓDIGO.

En las secciones anteriores se describieron las transformaciones que se pueden hacer a los programas sin que se modifique su funcionamiento: en los siguientes incisos se describirán las técnicas que se aplican al código para poder implantar las transformaciones. Estas técnicas permiten que se pueda analizar el código desde dos puntos de vista: local y global. El fundamento de estos métodos es el código de tres direcciones; con este código se puede dividir todo programa en bloques básicos, y teniendo bloques básicos es posible representar el programa por medio de gráficas dirigidas acíclicas. Con ayuda de dichas gráficas se implantan muchas de las transformaciones descritas. Otra ventaja de manejar a los programas a través de bloques es que se pueden implantar los criterios de optimización (de manera global o local) logrando con ésto tener la base de los algoritmos que mejoran código (en especial el generado por un compilador).

En esta sección se explican la naturaleza y construcción del código de tres direcciones, bloques básicos y representación de código de programas como gráficas.

CÓDIGO DE TRES DIRECCIONES.

Se llama **código de tres direcciones** a aquel cuyas instrucciones están formadas por a lo más tres operandos (de aquí el nombre de código de tres direcciones).

En las instrucciones en código de tres direcciones los operandos son sencillos, esto es, los operandos son variables o constantes (y no variables con índices como $A[i]$). Las instrucciones en código de tres direcciones pueden ser de los siguientes tipos:

- Instrucciones donde se asigna una expresión que tiene dos variables, constantes o nombres y una operación binaria y son de la forma : $X := y\,OP\,z$ (donde OP es una operación aritmética o lógica). Por ejemplo : $A := B + C$.

- Instrucciones donde se asigna una expresión que tiene una variable, constante o nombre y una operación unaria y son de la forma $X := OP\ y$ (donde OP es una operación unaria como la negación). Por ejemplo : $A := -B$.

- Instrucciones con las que se copia constantes o el contenido de variables de la forma $X := Y$. Por ejemplo: $A := 5$.

• Instrucciones donde el flujo del programa tiene un salto sin que antes tenga que cumplirse una condición y son de la forma $GOTO\ E$ (donde "E" es una etiqueta que indica a que lugar se dirige el flujo de ejecución de instrucciones).

• Instrucciones con las que puede haber un salto en el flujo de ejecución del programa, si es que se cumple una condición y son de la forma IF **OPLOG** $GOTO\ E$ donde **OPLOG** es una operación lógica de la forma $x\ OP\ y$, donde OP puede ser $<$, $>$, $=$, $\leq$, $\geq$, AND, OR y NOT.

• Instrucciones que llaman a ejecutar un procedimiento y son de la forma $PROC\ X_1$, donde X_1 pude ser uno o más parámetros, y $PROC$ es algún procedimiento.

• Asignaciones con índices como $X := A[i]$

• Instrucciones que asignan direcciones o son apuntadores de la forma $X := \hat{}Y$;

Todo código se puede representar como un código de tres direcciones (ver [Aho,et al, 1990]). Muchos compiladores después del análisis sintáctico y semántico, generan un código intermedio que es de tres direcciones y los compiladores que son optimizadores usan este código para optimizar y generar el código final. Por estas razones y porque es muy común que todo compilador genere un programa objeto escrito en un lenguaje ensamblador (que es un tipo de código de tres direcciones), es por lo que para muchos algoritmos y técnicas de optimización de código se toma como base y ejemplo al código de tres direcciones.

2.2.1.- BLOQUES.

Un primer análisis del código es el flujo de ejecución de instrucciones; al hacer esto se puede dividir el programa, agrupando conjuntos de instrucciones donde no hay saltos o modificación en el orden de ejecución del código. Esta división es llamada agrupación del código en bloques básicos. Un **bloque básico** es un grupo de instrucciones en el que la secuencia de ejecución es la misma que el orden de aparición de cada instrucción, esto es, el control de flujo entra al principio del bloque y sale al final sin detenerse y sin posibilidad de saltar excepto al final del bloque.

Ejemplo 2.10 :

El siguiente código sí es un bloque básico

$$(1^*)\ X := Y + Z$$

$$(2^*)\ W := X * Y$$

$$(3^*)\ Z := 17$$

$$(4^*)\ X := A[i]$$

Aquí no existen posibles desviaciones en el orden ejecución del código (no hay saltos o llamadas a otro fragmento de código), es por eso que este es un ejemplo de un bloque básico.

Ejemplo 2.11 :

El siguiente código no es un bloque básico:

$$(1)\ X := Y + Z$$

$$(2)\ W := X * Y$$

$$(3)\ IF\ W = 7\ GOTO\ (5)$$

$$(4)\ Z := 17$$

$$(5)\ X := A[i]$$

En la instrucción (3) el orden de ejecución del código se puede alterar al dar un salto a (5).

Para poder identificar a un bloque básico se localiza su primer elemento, con esto ya se pueden encontrar todos los bloques en un programa porque los primeros elementos marcan el final de un bloque anterior. Los primeros elementos de un bloque se llaman **líderes**.

Con estas ideas se tienen los elementos para construir un algoritmo que divida a todo código en bloques básicos. (Este algoritmo usa una variable N que identifica al número de líder que aparece en un código).

Algoritmo (2.1) Formación de bloques de código de tres direcciones.

1.- Hacer $N = 1$.

2.- Mientras exista código no agrupado en un bloque, tomar una instrucción y verificar si es líder de bloque de acuerdo a los siguientes criterios:

a) La primera instrucción del código es un líder (el líder 1).

b) Si la instrucción es el destino de cualquier salto en el flujo del programa entonces incrementar N en uno ($N <- N + 1$), marcar la instrucción como el líder N y pasar a **2.**

c) Si la instrucción está después de una instrucción de salto entonces incrementar N en uno, marcar la instrucción como el líder N y pasar a **2.**

3.- Formar el bloque como sigue:

a) Si N es distinto de 1, entonces guardar en un conjunto (llamado $bloque_{N-1}$) todas las instrucciones que estén desde el líder $N-1$ hasta antes del líder N.

Ejemplo 2.12 :

Dividir en bloques básicos el siguiente código:

(1) $A := 17$

(2) $B := B * A$

(3) $C := D - E$

(4) $E := G/H$

(5) $IF\ E = A\ GOTO$ (3)

(6) $G := X$[i]

(7) $IF\ C < D\ GOTO$ (8)

(8) $F := 17$

(9) $GOTO$ (3)

(10) $A := B$

Paso 1.- $N = 1$.

Paso 2a.- La primera instrucción (1) es el líder $N = 1$.

Paso 2b.- La instrucción (3) es el líder $N = 2$.

Paso 3a.- El $bloque_{N-1=1}$ se forma por las instrucciones (1) y (2).

Paso 2b.- La instrucción (6) es el líder $N = 3$.

Paso 3a.- El $bloque_{N-1=2}$ se forma por las instrucciones (3), (4) y (5).

Paso 2b ó 1c .- La instrucción (8) es el líder $N = 4$.

Paso 3a.- El $bloque_{N-1=3}$ se forma por las instrucciones (6) y (7).

Paso 2c.- La instrucción (10) es el líder $N = 5$

Paso 3a.- El $bloque_{N-1=4}$ se forma de las instrucciónes (8) y (9).

Paso 2c.- La instrucción (10) es el líder $N = 6$

Paso 3a.- El $bloque_{N-1=5}$ se forma de la instrucción (10) nada mas.

REPRESENTACIÓN DE BLOQUES BÁSICOS

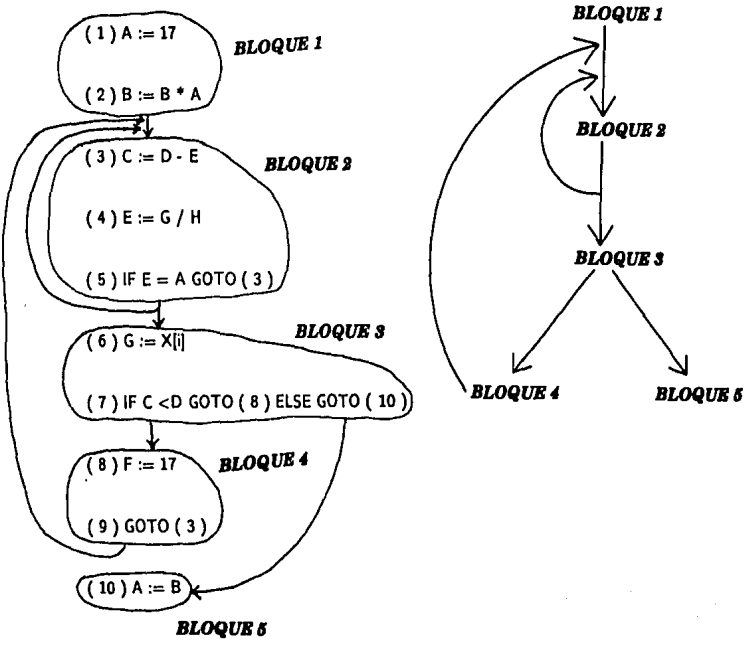
Aplicando el algoritmo para formar bloques (y previamente traduciendo los programas a un código de tres direcciones) se puede llegar a ver todo código como una **gráfica dirigida**[14]. En la representación de código como gráfica dirigida los nodos son bloques y las aristas dirigidas son el flujo del programa. Con esto, el problema de hacer que el código sea óptimo se traduce a

[14] Una **Gráfica dirigida** es un modelo matemático que consta de **nodos** y **aristas**, cada nodo puede estar aislado o unido a otro nodo por al menos una arista, y cada arista tiene una dirección. Se representa una gráfica dirigida como $G : [\, N, A \,]$ donde N es el conjunto de nodos y A es el conjunto de aristas.

optimizar las gráficas dirigidas (hacer que el recorrido en una gráfica dirigida sea el más corto). Muchos de los algoritmos para optimizar tienen que ver con teoría de gráficas [Aho- 1979].

Ejemplo 2.13 :

Mostrar la gráfica de flujo para el código del ejemplo 2.12:



Existen muchos tipos de gráficas que representan programas y de estos tipos hay casos en que las aristas que parten del final de un nodo llegan al principio del mismo nodo de origen, cuando ocurre esto se dice que existe un **lazo**.

LAZOS.

Un **lazo** en una gráfica es un conjunto de nodos tales que entre cualquier pareja de nodos existe al menos una arista que los une o un camino, esto quiere decir que todos los nodos están fuertemente conectados.

Así pues, un lazo en una gráfica de flujo es un conjunto de nodos tales que están fuertemente conectados y tienen entrada única. Del ejemplo 2.13 existe el lazo formado por el bloque 2 y el lazo formado por los bloques 2, 3 y 4.

Ejemplo 2.14 :

El siguiente bloque forma un lazo (es el bloque 2 del ejemplo, 2.12)

(3) C := D - E

(4) E := G / H

(5) IF E = A GOTO (3)

Dentro de un lazo puede haber otros lazos y si en un lazo no existen otros lazos se dice que es un lazo interno. El ejemplo 2.14 es un lazo interno y los bloques 2, 3 y 4 forman un lazo que contiene un lazo interno (en el ejemplo 2.13 el bloque 2).

Se puede mejorar el código haciendo que el control de flujo en un lazo sea de rápida ejecución, para lograr esto se emplean ciertas técnicas como la eliminación de código inactivo. El uso de lazos para optimizar puede lograr que estructuras como las recursivas ocupen poco espacio o sean de ejecución veloz.

2.2.2.- GRÁFICAS.

Otro tipo de estructura útil que se construye a partir del código de tres direcciones son las **gráficas dirigidas acíclicas** (llamadas "GDA") y esta construcción surge de manera natural porque el código de tres direcciones es una representación linealizada de una GDA [Aho. 1990]. Las GDA's se aplican para analizar de manera individual los elementos que forman los bloques básicos.

La GDA tiene las siguientes características y la siguiente información :

- Los nodos internos de la gráfica representan a los operadores.

- Los nodos sin hijos (a los que llamaremos hojas[15]) representan a los operandos de las expresiones y para distinguir estos nombres de otros tienen un índice.

- Los nodos que tienen los operadores también tendrán los nombres de las variables a las que son asignadas las expresiones y éstas se diferencian de las variables representadas por las hojas porque aquí no tendrán índices.

- En una GDA el ancestro de una hoja es el nodo que representa una operación (asociado con una o más varibles) al que está unido. Al nodo sin ancestros se llama raíz y es un nodo cuyas variables asignadas no son utilizadas como elementos en otras instrucciones.

Intuitivamente el algoritmo para hacer un GDA es el siguiente: teniendo en el bloque sólo expresiones se crea un nodo para cada nuevo operando u operador. En los nodos estarán las operaciones u operadores y se marcan con el nombre de la variable a la que se asigna la expresión u operación. Las hojas serán los operandos y tendrán los nombres de las variables involucradas en la operación. Si las variables no han sido usadas serán escritas con un índice. Un nodo puede ser hoja (operando) de otro nodo si es que en la expresión se vuelve a usar la variable correspondiente.

[15] La razón de llamar hojas a los nodos sin hijos de estas gráficas es porque las GDA's tienen una representación equivalente en forma de árbol; dichas estructuras son los árboles sintácticos y en ellos corresponden a los nodos sin hijos de las GDA's. La razón de usar las gráficas en lugar de los árboles es porque una GDA proporciona la misma información del árbol pero de forma más compacta mostrando además de manera evidente toda subexpresión común [Aho- 1990]. Una subexpresión común es toda aquella expresión aritmética que se repite más de una vez en un código y se llama subexpresión porque puede aparecer como parte de otras instrucciones.

Algoritmo (2.2) Construcción de una GDA

Dado el código de tres direcciones en un bloque :

1.- Crear una hoja marcada en su interior con el operando (o elemento a asignar) si es que este nodo no ha sido definido.

2.- Crear un nodo con el operador y unirlo con el (los) operador (es) que le corresponde(n) a la instrucción. En caso de que la instrucción no tenga operadores nada se hace.

3.- Asociar al nodo el nombre de la variable o identificador al que se asigna la expresión.

Ejemplo 2.15 :

Sea el siguiente bloque:

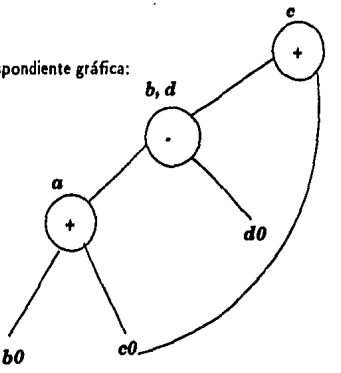
(1) $a := b + c$;

(2) $b := a - d$;

(3) $c := b + c$;

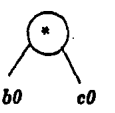
(4) $d := a - d$;

Con su correspondiente gráfica:

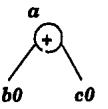


Por el paso (1) se crean las hojas (b_0) y (c_0).

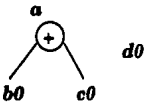
Por el paso (2) se crea el nodo (+) y se une a las dos hojas.



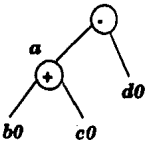
En el paso (3) se asocia al nodo (+) la variable (a).



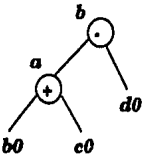
Por el paso (1) sólo se crea la hoja (d_0).



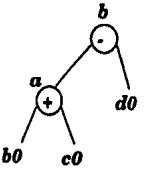
En el paso (2) se crea el nodo (−) y se asocia a la hoja (d_0) y al nodo (a).



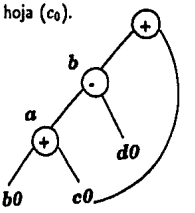
En el paso (3) se asocia al nodo (−) la variable (b).



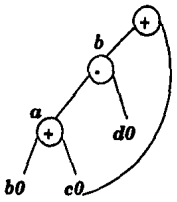
Por el paso (1) no se crea ninguna hoja porque ya existen (b) y (c).



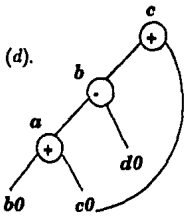
En el paso (2) se crea el nodo (+) y se une al nodo (b) y a la hoja (c_0).



Por el paso (3) se asocia al nodo (+) la variable (c).

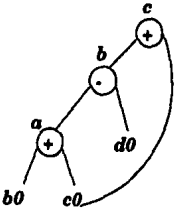


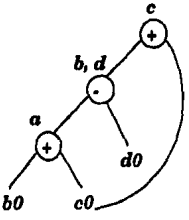
Por el paso (1) no se crean hojas porque ya existen (a) y (d).



En el paso (2) tampoco se crea nada porque ya existe el nodo (−) que une a las hojas (a_0) y (d_0).

En el paso (3) se asocia al nodo (−) la variable (d).





Las GDA además son útiles para el análisis y optimización del código de bloques básicos; muchas transformaciones que no modifican a los programas se construyen a partir de una GDA.

GENERACIÓN DE CÓDIGO A PARTIR DE UNA GDA.

El código de tres direcciones es una representación lineal de una GDA ya que cada elemento de la gráfica corresponde a un elemento del código de tres direcciones: cada hoja de la gráfica corresponde a un operando, variable o constante y los nodos son los operadores. Así se tiene que para **instrucciones simples**, como $A := B * C$, la representación en una GDA es también simple[16]. Para instrucciones simples la GDA consta nada más de un nodo y uno o dos hijos nada más, así que la transformación de la gráfica al código es inmediata y casi por definición como se ve en el siguiente algoritmo.

Algoritmo (2.3) Generación de código de GDA's simples

Entrada: GDA simple (formada de un bloque).

Salida: código de tres direcciones.

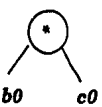
1.- Escribir el nombre con que está marcado el nodo, seguido del símbolo de asignación.

2.- Escribir el nombre del operador que tiene el nodo.

3.- Escribir el nombre de la hoja izquierda como el operando izquierdo y el nombre de la hoja derecha como el operando derecho.

Ejemplo 2.16 :

De la siguiente GDA generar el código de tres direcciones correspondiente:



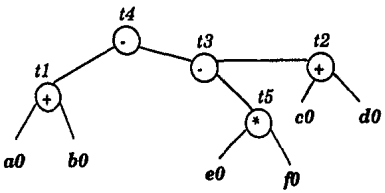
A esta gráfica le corresponde el siguiente código: $a := b * c$

[16] En un código de tres direcciones se dice que una instrucción es simple si se trata de la asignación de una expresión aritmética como $A := B + C$, o de una variable o constante (por ejemplo $A := B$).

Este es un ejemplo sencillo en el que puede observarse la transformación inmediata de la gráfica al código (el nodo es la operación y las hojas son los operandos), pero existen GDA's muy complicadas donde el paso no es claro:

Ejemplo 2.17 :

Sea la siguiente GDA:



Lo fundamental en este tipo de gráficas es saber a partir de cuál nodo se empieza a generar el código; otro problema es conocer los valores de las hojas para cada nodo que se quiera traducir, además existe la dificultad de saber qué orden se debe seguir para traducir la GDA a código. Si se tiene un orden para evaluar las hojas y los nodos, la tarea de generar las instrucciones se reduce al caso sencillo del ejemplo 2.16, porque una instrucción de tres direcciones es equivalente a un nodo con sus respectivas hojas evaluadas.

Así, lo más importante para generar código a partir de una GDA es llevar el orden correcto para la evaluación de hojas y el recorrido de los nodos de la gráfica. El algoritmo de recorrido de nodos debe asegurar que no se altere el sentido del programa original de donde se obtuvo la GDA.

Un algoritmo para generar código a partir de una GDA básicamente definirá un orden de recorrido para la evaluación de las hojas y nodos. El siguiente algoritmo propuesto por Sethi, Aho y Ullman [Aho et al 1990] hace que el recorrido de los nodos sea de tal manera que sólo se genere código para un nodo, cuando su arista más a la izquierda haya sido evaluada. Este algoritmo usa una variable (que llamaremos X) para numerar y ordenar los nodos de la gráfica, y una variable (que llamaremos N) para representar un nodo de la gráfica.

Algoritmo (2.4) Recorrido de nodos en una GDA

Entrada: GDA.

Salida: orden de recorrido para los nodos de la gráfica.

1.- Hacer $X = 0$;

2.- Mientras queden nodos sin marcar:

2.1.- Seleccionar un nodo N no marcado tal que todos sus padres ya se marcaron o que no tenga padres por marcar.

2.2.- Hacer $X = X + 1$.

2.3.- Marcar el nodo N con el número X.

2.4.- Mientras el hijo de más a la izquierda del nodo N (llamémosle Z) no tenga padres marcados y no sea una hoja:

2.4.1.- Hacer $X = X + 1$.

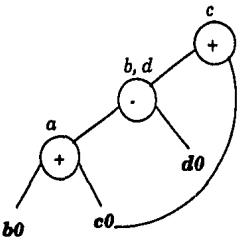
2.4.2.- Marcar Z con el número X.

2.4.3.- Hacer $N = Z$

3.- El recorrido de los nodos es el inverso al orden marcado en cada nodo.

Ejemplo 2.18 :

Sea la GDA:



De acuerdo al algoritmo tenemos los siguientes pasos:

- $X = 0$ (paso 1).
- $N = C$ (paso 2.1).
- $X = 1$ (paso 2.2).
- Se marca C con 1 (paso 2.3).
- Z es el nodo con nombre b, d (paso 2.4).
- $X = 2$ (paso 2.4.1).
- Se marca al nodo b, d con 2 (paso 2.4.2).
- $N =$ nodo b, d (paso 2.4.3).
- Z es el nodo con nombre a (paso 2.4).
- $X = 3$ (paso 2.4.1).

- Se marca al nodo a con 3 (paso 2.4.2).

- **N** = nodo a (paso 2.4.3).

- Como la hoja b_0 tiene padres marcados, termina el ciclo (paso 2.4).

- El recorrido es el orden decreciente de los números con que se marcaron los nodos y es: 3, 2, 1, esto es a, (b,d) y c (paso 3).

La aplicación del algoritmo del recorrido da un orden para la obtención del código y se puede aplicar el algoritmo 2.3 de generación de código a cada nodo porque está garantizado que cada nodo tiene en sus aristas hojas o un nodo cuyo código ya se desarrolló.

Siguiendo el orden del recorrido de nodos dado en el ejemplo anterior (ejemplo 2.18) y aplicando el algorimo 2.3 (de generación de código) a cada nodo se tiene el siguiente código:

(1) $a := b + c$

(2) d, $b := a - d$

(3) $c := d$, $b + c$

En este código el nombre de los operandos no aparece con el subíndice 0 (por ejemplo b_0) debido a que esta marca es solamente para distinguir en una GDA si el nombre de la variable corresponde a un nodo o una hoja (las variables con subíndice corresponden a hojas). También en este código la variable que aparece escrita como d, b representa al nodo marcado con las variables b y d[17], y para sustituir d, b por una sola variable se elige a una variable (b o d) y se escribe en lugar de b, d; si se elige a b entonces el código queda así:

(1) $a := b + c$

[17] Los nodos marcados en una GDA con más de un nombre de variable representan subexpresiones comunes

(2) $b := a - d$

(3) $c := b + c$

En un nodo marcado con el nombre de muchas variables para elegir cual de todas ellas se escribe se revisa a los siguientes bloques de todo el programa: si no se usa ninguna de las variables se elige una al azar, pero en el caso de que en el programa se utilice más de una variable del nodo en más de un bloque, en la traducción de GDA a código se añade una proposición que copie el valor de una en otra.

Ejemplo 2.19 :

Si en el código traducido del ejemplo 2.18 las variables b y d (con las que está marcado un nodo) se utilizan más de una vez en el resto del programa, entonces el código traducido queda así:

(1) $a := b + c$

(2) $b := a - d$

(3) $c := b + c$

(4) $d := b$

Para facilitar la tarea anterior, se presenta el siguiente algoritmo para elegir cual variable se escribe de un nodo con muchos nombres.

Algoritmo (2.5) Elección de variables en nodos

Entrada: Código traducido de una GDA que tiene nodos con más de un nombre.

Salida: Código traducido de una GDA donde solamente aparece un nombre por cada nodo.

1.- Mientras exista un nodo (con más de un nombre) sin marcar.

1.1.- Localizar un nodo con más de un nombre y marcarlo.

1.2.- Para cada nombre de variable que tenga el nodo:

1.2.1.- Comparar los nombres de las variables de las siguientes instrucciones (del bloque o GDA asociada) con cada nombre de variable del nodo.

1.2.2.- Marcar cada nombre de variable del nodo que se repita más de una vez en el resto del programa.

1.2.3.- Si hay solamente una varible marcada en el nodo, entonces utilizar esta variable para asignarle la expresión que representa todo el nodo.

1.2.4.- Si no, (en el nodo existe mas de una variable marcada) elegir una variable al azar, asignarle la expresión correspondiente al nodo y escribir el código de asignación de la variable elegida a cada una de las restantes

variables marcadas.

La aplicación de este algoritmo se dió en el ejemplo 2.19.

2.2.3.- TIPOS DE OPTIMIZACIÓN DE CÓDIGO.

Con las estructuras descritas se tiene una manera de estudiar el código: se puede ver como una gráfica constituida por bloques de instrucciones y a su vez cada bloque se representa como gráfica (GDA); con estos elementos y las transformaciones que modifican al código se tiene la base para poder enunciar los algoritmos que optimizan el código dado por un compilador. El fundamento de estos algoritmos es aplicar transformaciones que mejoren el código de manera local (esto es, optimizar las instrucciones de los bloques básicos) y de manera global (es decir, mejorar las gráficas formadas con nodos que son bloques básicos).

CAPITULO TRES

3.- OPTIMIZACIÓN.

3.0.- INTRODUCCIÓN.

En este capítulo se describe cómo hacer para que el código generado por un compilador sea lo mejor posible (más rápido y pequeño).

Teniendo las principales transformaciones y criterios para mejorar todo programa se usan las técnicas que permiten representar todo código en tres direcciones y en bloques, para crear los algoritmos que mejoren el código generado por los compiladores.

Los algoritmos para optimizar nacen directamente de los criterios y transformaciones analizadas en el capítulo dos. Los métodos que permiten el análisis de todo código hacen que se tengan dos puntos de vista de cualquier programa: local y global[18], con estas técnicas se infiere que los algoritmos para optimizar resultan de la aplicación de transformaciones y métodos para mejorar código de manera local y global.

Optimizar formalmente, es la aplicación de métodos para determinar los valores de las variables que hacen máximo el rendimiento de un proceso o sistema. Para el proceso de compilación ya se definió al tamaño y tiempo de ejecución del código traducido como las variables a optimizar.

La optimización es el algoritmo (conjunto de pasos) que resuelve la tarea de optimizar cierto sistema. La existencia del algoritmo depende de que el problema que se resolverá tenga solución. El generar código óptimo es un problema indecidible debido a las variables para la optimización (rapidez y tamaño), dado que no siempre el programa más rápido ocupa menos espacio y no todo código pequeño es el más veloz[19].

Para evitar la indecidibilidad en la optimización de código traducido el problema se divide; se aplican métodos para modificar el programa compilado tales que cuando se reduce el tamaño del código no lo hace más lento y viceversa. Una característica de estas técnicas es que su

[18] Esto es porque todo código se puede ver como una gráfica formada por bloques de instrucciones (punto de vista global) y a su vez cada bloque se representa como una GDA (punto de vista local).

[19] La demostración de que el problema de optimización es indecidible fue hecha por A. Aho en 1970 [Aho-1970].

aplicación generará problemas con solución (y por lo tanto tendrán algoritmos). Los métodos que modifican el código compilado y hacen transformaciones deben evitar la alteración en el funcionamiento del programa.

3.1.- LOCAL.

Un primer método para aplicar los métodos y transformaciones que mejoran el código es dividir el programa en intervalos y aplicar la optimización a cada intervalo. A esta técnica se le llama **optimización por intervalos**. Un defecto de la optimización por intervalos es que al no precisar *"el tamaño del intervalo"* no se toma en cuenta las posibles estructuras de un programa para dividirlo y sucede que si el intervalo es grande los métodos de mejora son complicados y se pierde precisión, y por el contrario, si el intervalo es pequeño se pierde generalidad.

Gracias a las técnicas que permiten el análisis de código es posible dividir a todo programa según las estructuras lógicas que use (IF's, $WHILE$'s, etc). Cada estructura que aparece en el código representa el principio de un bloque básico, y se tendrán unidades o bloques básicos que se representan como GDA's; con estos elementos se pueden aplicar técnicas y transformaciones que hagan de todo programa un código lo más pequeño y veloz posible.

Los algoritmos de optimización local surgen de los criterios y transformaciones que mejoran de manera local a todo código y son los siguientes:

- Eliminación de subexpresiones comunes.
- Eliminación de código inactivo.
- Simplificación algebraica.

3.1.1.- ELIMINACIÓN DE SUBEXPRESIONES COMUNES.

Bajo el nombre de *eliminación de subexpresiones comunes locales* se agrupan los métodos dados en la sección (2.1.1) para sustituir expresiones ya calculadas en bloques básicos

La construcción de una GDA a partir de un bloque básico localiza las subexpresiones comunes, ya que cuando se trata de añadir un nuevo nodo a la gráfica si ya hay un nodo con los mismos hijos [22] entonces el nodo no se crea, solamente se asocia su nombre al nodo ya hecho [23]. Una subexpresión común se localiza fácilmente en una GDA ya que todo nodo marcado con más de un nombre tendrá una subexpresión común. [24]

Ejemplo 3.1 :

Sea el siguiente código tomado de la sección (2.1.1) para la eliminación de subexpresiones comunes. Supóngase que es un bloque básico.

(1) $a := b + c$

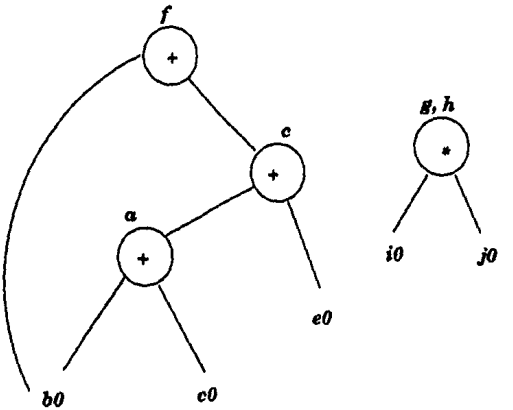
(2) $g := i * j$

(3) $c := a + e$

(4) $f := b + c$

(5) $h := i * j$

Con su correspondiente GDA:



[22] Un hijo en una GDA son los extremos de los nodos que tienen operaciones y una o más varibles asignadas. Los hijos pueden ser hojas u otros nodos

[23] Ver algoritmo (2.2) "Construcción de una GDA"

[24] Esto quiere decir que existe más de una variable a la que se le asignó el valor dado por una misma expresión

En esta gráfica el nodo correspondiente a "$*$" está marcado con dos variables ("g" y "h") debido a que la expresión $i*j$ se asigna a dos variables, sin embargo aunque también la expresión $b+c$ es asignada a dos variables ("a" y "f") en la GDA no aparece otro nodo marcado con dos variables, esto se debe a que en la segunda ocasión donde se emplea la expresión $b+c$ (en (4)) dicha operación no representa el mismo valor que en la primera vez que se usó (en (1)) porque el valor de c cambió debido a la instrucción $c{:=}a+e$ (instrucción (3))[25]. Por lo tanto el nodo $*$ representa la asignación a dos variables de un mismo valor dado por una misma expresión, es decir, una subexpresión común.

Para aplicar la eliminación de subexpresiones comunes se debe elegir en cuál de las variables se eliminará la expresión común. En este caso se eliminará h; y aplicando la eliminación se tiene:

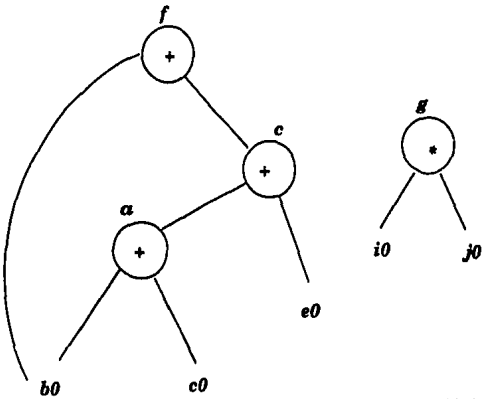
(1*) $a := b+c$

(2*) $g := i*j$

(3*) $h := g$

(4*) $c := a+e$

(5*) $f := b+c$



En este ejemplo se eliminó la expresión $(i*j)$ de la asignación a h (esto mejora la velocidad y tamaño). Pero para la elección de nodos con más de una variable se tiene un algoritmo (el algoritmo 2.5) que permite decidir si se puede prescindir de una variable al momento de pasar de una GDA a código. Supóngase que de este bloque la variable h no es usada en el resto del código, entonces aplicando el algoritmo para elegir variables en nodos se obtiene:

(1**) $a := b+c$

(2**) $g := i*j$

[25] Todos estos aspectos se reflejan en el algoritmo de construcción de una GDA (algoritmo 2.2).

(3**) $c := a + c$

(4**) $f := b + c$

En este caso se redujo el código, por lo tanto para la eliminación de subexpresiones comunes se necesita el algoritmo de elección de variables (algoritmo 2.5) para los nodos que contienen subexpresiones comunes.

ALGORITMO PARA LA ELIMINACIÓN DE SUBEXPRESIONES COMUNES LOCALES.

Con lo anterior, ya se tienen los elementos necesarios para eliminar las subexpresiones comunes de un bloque básico:[26]

Algoritmo (3.1) Eliminación de subexpresiones comunes locales

Entrada: Código de tres direcciones.

Salida: Código del que se ha eliminado toda subexpresión común local (es decir, no existen subexpresiones comunes en los bloques básicos que se definen en cada programa).

* **1.**- Dividir el programa en bloques básicos (algoritmo 2.1) [27] .

* **2.**- Para cada bloque

    * **2.1.**- Transformar el bloque a una GDA (algoritmo 2.2).

    * **2.2.**- Transformar la GDA a código

[26] Este es el primer algoritmo cuya función es optimizar el código generado por un compilador y es por eso que para su construcción se hace uso de muchos de los conceptos dados anteriormente; en general varios de los algoritmos para mejorar código se componen de muchos "subalgoritmos".

[27] Para todos los demás algorimos de optimización local siempre se utilizará este paso.

* **2.2.1.-** Encontrar un orden de recorrido de los nodos (algoritmo 2.4).

* **2.2.2.-** Para cada nodo, (siguiendo el orden del recorrido)

* **2.2.2.1.-** Decidir que variables usar (sólo en nodos con mas de una variable)

(algoritmo 2.5).

* **2.2.2.2.-** Generar código a partir de la GDA (algoritmo 2.3)

para nodos con una variable.

En el caso de que en un nodo con muchas variables ninguna se pueda eliminar o se necesite usar más de una, el algoritmo para elección de variables (algoritmo 2.5) indica que se debe asignar la expresión del nodo a una variable y copiar dicha variable a las restantes; para estos casos el código de copia de una variable a otra se debe escribir siguiendo el criterio de propagación de copias (sección 2.1.6) que consiste en que para cada copia de la forma $A{:=}B$ se utilize en el resto del código la variable B en lugar de A. Esto se debe hacer así porque de esta manera se evita el manejo de muchas variables[28] y el desperdicio de memoria.

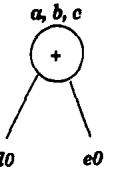
Ejemplo 3.2 :

Sea el siguiente bloque :

(1) $a := d + e$

(2) $b := d + e$

(3) $c := d + e$



[28] Como A y B tienen el mismo valor se pueden utilizar ambas, pero con este criterio se garantiza que nada más una se empleará en lo sucesivo. Esto es ventajoso también cuando hay "memoria cache".

Aplicando la eliminación de subexpresiones comunes (sin usar el criterio de propagación de copias) este bloque queda así:

(1*) $a := d + e$

(2*) $b := a$

(3*) $c := b$

No se elimina ninguna variable del nodo "+" porque el algoritmo de elección de variables sólo verifica el uso de cada variable del nodo en el bloque o GDA asociada. Ahora si se aplica el criterio de propagación de copias el código queda así:

(1**) $a := d + e$

(2**) $b := a$

(3**) $c := a$

Este criterio se usa porque deja de manera evidente el código inactivo y así es más fácil eliminarlo.

3.1.2.- ELIMINACIÓN DE CÓDIGO INACTIVO.

El objetivo de esta transformación es eliminar código que no se utiliza en el programa. El código inactivo puede surgir como resultado de la compilación o debido a la aplicación de transformaciones para optimizar el programa, por ejemplo, si del último bloque considerado (ejemplo 3.2) alguna de las variables de copia (b ó c) no se emplea en el resto del programa entonces se puede eliminar[29].

[29] De aquí la importancia de usar el criterio de propagación de copias. Si en el ejemplo anterior (ejemplo 3.2) no se hubiera empleado el criterio de propagación de copias y la variable b no se usa en el resto del código, entonces no se podría eliminar $b := a$ porque se emplea b en la instrucción $c := b$

ALGORITMO PARA LA ELIMINACIÓN DE CÓDIGO INACTIVO.

Un método formal para eliminar código inactivo utiliza a las GDA's; en estas gráficas el código inactivo son los nodos sin ancestros[30] cuyas variables no se usan más (las variables del nodo no están "activas").

En una GDA el nodo que representa a una instrucción cuyo valor no es empleado en otras instrucciónes es un nodo sin ancestros. Estos nodos pueden ser de dos tipos: uno es donde el nodo sin ancestros es el ancestro de más de un nodo (en el ejemplo 3.1 el nodo + marcado con la variable f es un ejemplo de ellos); y el otro tipo lo son los nodos aislados que son nodos sin ancestros y que a su vez ellos no son ancestros de algún otro nodo (en ejemplo 3.1 el nodo * marcado con las variables g, h es un nodo de este tipo). En ambos casos este tipo de nodos son la representación de un valor que no se usa más en el bloque, y este valor se asigna a cada variable que marca al nodo. Entonces si dicho valor no es usado en el resto del código, las variables que lo representan se pueden eliminar por inactivas; pero si el valor es asignado a más de una variable y es utilizado en más lugares fuera del bloque, entonces se pueden eliminar las copias inecesarias del mencionado valor. En estos criterios se basa el siguiente algoritmo:

Algoritmo (3.2) Eliminación de código inactivo de bloques básicos.

Entrada: Bloques básicos (código de tres direcciones) con sus correspondientes GDA's.

Salida: Un programa sin código inactivo.

* **1.**- Mientras exista un bloque sin marcar:

    * **1.1.**- Elegir un bloque no marcado, marcarlo y para la GDA correspondiente:

    * **1.2.**- Localizar cada nodo sin ancestros y marcarlo.

* **2.**- Mientras exista un nodo marcado en todo el código (no solamente en un bloque):

    * **2.1.**- Siguiendo el control de flujo del código, elegir el primer nodo marcado, y con dicho nodo:

[30] Ver secc. 2.2.

*** 2.2.-** Comparar cada hoja de las siguientes GDA's de cada bloque (siguiendo el control de flujo a partir del nodo elegido), con la(s) variables (no las hojas) asociadas al nodo elegido, contar el número de veces en que cada variable(s) del nodo es igual a una hoja.

*** 2.3.-** Si ninguna hoja coincidió con la(s) variables del nodo, entonces marcar dicho nodo *como código inactivo*, si no, entonces: Marcar como *código inactivo* solamente la(s) variable(s) del nodo que no haya(n) coincidido alguna vez con una hoja.

*** 2.4.-** Quitar la marca del nodo elegido.

*** 3.-** Eliminar los nodos y nombres marcados como código inactivo.

La eliminación de código inactivo en bloques generalmente se aplica después de la eliminación de subexpresiones comunes locales, porque el uso del criterio de propagación de copias permite convertir expresiones de copia en código inactivo.

Ejemplo 3.3 :

Considérese el bloque :

(1) $A := B + C$

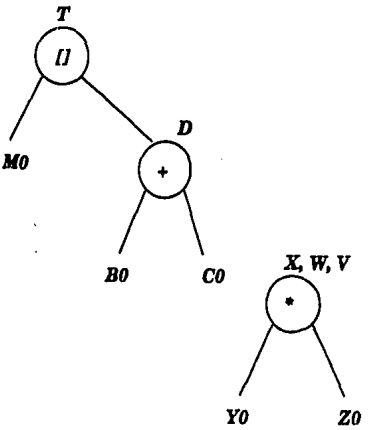
(2) $X := Y * Z$

(3) $T := M[\ A\ ]$

(4) $D := B + C$

(5) $W := Y * Z$

(6) $V := Y * Z$



En este ejemplo los nodos sin ancestros son: el que tiene la operación "[]" (con la variable asociada T) y el nodo"$*$" (con las variables asociadas X, W y V). De aquí se considerar los siguientes casos:

a) cuando la variable de "[]" (que es T) está activa y las variables de "$*$" no lo están;

b) cuando T está activa y uno o más variables de "$*$" (X, V ó W) están activas;

c) cuando T no está activa y alguna de las variables de "$*$" están activas;

d) y por último cuando todas las variables de ambos nodos no están activas.

Para el caso a) cuando T está activa y todas las variables de "$*$" están inactivas, el algoritmo en este bloque de ejemplo funciona así:

Se marcan los nodos "[]" y "$*$" (pasos 1...).

Se marca el nodo "$*$" como inactivo (pasos 2...).

Eliminar el nodo "$*$" (paso 3).

Para el caso b) donde la variable del nodo "[]" está activa y uno o más variables de "$*$" están activas, el algoritmo funciona así:

Se marcan los nodos "[]" y "$*$" (pasos 1...).

Marcar como código inactivo sólo las variables del nodo "$*$" que no estén activas, si todas las variables de "$*$" están activas entonces ningún nodo ("[]" y "$*$") se marca como inactivo (pasos 2...).

Eliminar solamente las variables marcadas como inactivas (paso 3).

Para el caso *c*) donde T es inactiva y alguna de las variables del nodo "*" (X ó V ó W) son activas, el algoritmo en este bloque de ejemplo funciona así:

Se marcan los nodos "[]" y "*" (pasos 1...).

Se marca el nodo "[]" como inactivo (pasos 2...) y eliminarlo (paso 3).

Para cada variable del nodo "*" que sea inactiva marcarla como código inactivo (pasos 2...).

Eliminar el nodo "[]" y todas las variables del nodo "*" que estén marcadas como inactivas (paso 3).

Para el caso *d*) donde todas las variables de los nodos sin ancestros están inactivas, el algoritmo hace lo que sigue:

Se marcan los nodos "[]" y "*" (pasos 1...).

Se marcan los nodos "[]" y "*" como inactivos (pasos 2...).

Se eliminan los nodos "[]" y "*" (paso 3).

Con estas transformaciones se mejora el código de bloques respecto al uso de las variables. Existen algoritmos de optimización que mejoran o reducen expresiones algebraicas.

3.1.3.- SIMPLIFICACIÓN ALGEBRAICA.

En los programas, generalmente aparecen instrucciones matemáticas por lo que no son raras las expresiones algebraicas en los códigos. En el capítulo dos se presentaron las técnicas y transformaciones en expresiones algebraicas que mejoran al código y no modifican el funcionamiento del programa; estas transformaciones se pueden expresar algorítmicamente y dichos algoritmos en su conjunto simplifican algebraicamente el código de los bloques básicos.

Debido a su relación con el criterio de transformación algebraica de la sección 2.1.3 los algoritmos que simplifican algebraicamente se dividen en *eliminación de identidades algebraicas, reducción de intensidad y cálculo previo de constantes.* En conjunto forman el algoritmo de simplificación algebraica.

ALGORITMO DE ELIMINACIÓN DE IDENTIDADES ALGEBRAICAS.

El objetivo de este algoritmo es eliminar del código todas las expresiones algebraicas que son identidades para la suma, resta, multiplicación y división.

Algoritmo (3.3) Eliminación de identidades algebraicas.

Entrada: Un bloque básico.

Salida: Un bloque sin identidades algebraicas expresadas en su código.

*** 1.-** Para cada instrucción en la que existe alguna suma (símbolo '+'), resta ('−'), multiplicación ('*') o división ('/') :

Si aparecen expresiones de la forma $X + 0$, ó $0 + X$, ó $X - 0$, ó $X * 1$, ó $X/1$ sustituir en cada caso la expresión algebraica por la expresión X.

El algoritmo elimina los elementos que son identidad para la suma y resta (el cero), y para la multiplicación y división (el uno) de acuerdo a los criterios de la sección 2.1.3.

Con este algoritmo se reduce el tamaño y el tiempo de ejecución del código. Ejemplos de la aplicación de este algoritmo son los dados en la sección 2.1.3.

ALGORITMO PARA LA REDUCCIÓN DE INTENSIDAD EN BLOQUES.

Otra optimización del código de expresiones algebraicas es la reducción de intensidad que consiste en sustituir operaciones "complicadas" por operaciones "sencillas". Antes de escribir el algoritmo se mencionan de nuevo los criterios: cambiar potenciación por multiplicación, multiplicación por sumas, división por multiplicación. Este algoritmo pretende que el código sea de rápida ejecución por lo que el tamaño del programa puede crecer después de su aplicación.

Algoritmo (3.4) Reducción de intensidad en bloques básicos.

Entrada: Un bloque básico.

Salida: Un bloque básico donde todas las expresiones algebraicas son las más rápidas de ejecutar (esto es, se redujo la intensidad de las operaciones algebraicas).

*** 1.-** Para cada expresión algebraica:

Sustituir toda exponenciación con exponente entero positivo, por código de multiplicación, escribiendo el producto de la base tantas veces como lo indique el exponente, es decir, expresiones de la forma a^n se sutituyen por el producto: $a.a.a \ldots .a$ (el producto de a *n-veces*)[31].

Sustituir toda expresión de multiplicación donde al menos un factor sea un entero positivo, por instrucciones de suma; se escriben tantas sumas como el entero positivo lo indique y cada instrucción es la suma del otro elemento consigo mismo. Expresiones de la forma $m.n$ se sustituyen por la suma: $m + m + \ldots + m$ (la suma de m *n-veces*).

Sustituir toda división por la multiplicación del dividendo por el inverso del divisor. Expresiones como a/b son sustituidas por la multiplicación: $a * (1/b)$ (donde $b \neq 0$).

[31] Estas sustituciones y todas las que se mencionen en este algoritmo se escriben en código de tres direcciones

Los ejemplos de la aplicación de este algoritmo están en la sección 2.1.3.

ALGORITMO PARA ELIMINAR EXPRESIONES PREVIAMENTE CALCULADAS.

La última operación para la mejora de expresiones algebraicas es el cálculo previo de constantes que consiste en sustituir operaciones algebraicas por la constante que calculan[32].

Algoritmo (3.5) Cálculo previo de constantes.

Entrada: Un bloque básico.

Salida: Un bloque básico donde existen constantes en lugar de las expresiones algebraicas que las calculan.

* **1.-** Para cada expresión algebraica:

Si en una operación binaria los dos operadores son constantes, sustituir la operación por el resultado del cálculo.

Este algoritmo reduce el tamaño del código y la velocidad de ejecución, porque una constante ocupa menos espacio que una expresión algebraica equivalente, y la evaluación de dichas expresiones ocupa mas tiempo que el uso del valor que representan.

Ya descritos los diversos algoritmos para simplificar los programas en sus expresiones algebraicas se puede definir un algoritmo general de simpificación algebraica.

ALGORITMO PARA LA SIMPLIFICACIÓN ALGEBRAICA.

El siguiente algoritmo es una combinación de los algoritmos que se refieren a la optimización de expresiones algebraicas[33].

[32] Por ejemplo $X := 2 * 3$ se sustituye por $X := 6$. (Ver sección 2.1.3)

[33] Cada algoritmo de simplificación algebraica se puede aplicar de forma individual.

Algoritmo (3.6) Simplificación algebraica en bloques básicos.

Entrada: Código generado por un compilador (ya dividido en bloques básicos para su análisis).

Salida: Un programa cuyo código tiene toda expresión algebraica simplificada.

* **1.**- Para cada bloque básico:

    * **1.1.**- Eliminar las identidades algebraicas (algoritmo 3.3).

    * **1.2.**- Reducir la intensidad de las operaciones algebraicas (algoritmo 3.4).

    * **1.3.**- Usar el cálculo previo de constantes para cada expresión (algoritmo 3.6).

Este algoritmo se construyó en base de otros algoritmos, y de igual manera, para optimizar el código de todo bloque básico, el algoritmo a seguir es aplicar cada uno de los algoritmos dados en esta sección. En este algoritmo cada instrucción es un algoritmo de este capítulo, y el orden de las instrucciones es el orden en que se presentaron en esta sección.

3.2.- GLOBAL.

El problema de obtener programas compilados cuyo código sea rápido y pequeño sólo se soluciona en parte al aplicar transformaciones en los bloques básicos. Con la optimización local sólo se obtienen programas eficientes "a trozos", es decir sólo en los bloques.

No se puede olvidar que el código generado por un compilador es un todo cuya representación es una gráfica con nodos hechos por bloques básicos. Entonces para la optimización completa de los programas se necesita mejorar de manera global el código[33] .

Para optimizar globalmente se utilizan algoritmos basados en las técnicas y transformaciones expuestas en el capítulo 2:

- Eliminación de subexpresiones comunes.
- Propagación de copias.

Y un método nuevo:

- Optimización en lazos.

Las transformaciones globales no son un sustituto de las locales, ambas deben aplicarse porque unas son el complemento de las otras.

Una diferencia entre la optimización local y la global es que en la local se analizan y aplican transformaciones a un conjunto de instrucciones en que el control de flujo es lineal o directo (no hay saltos debidos a *GOTO's*, por ejemplo), mientras que para la global pueden presentarse "saltos" que provocan que el control de flujo siga diferentes caminos. De estas observaciones surge la necesidad de definir algunos conceptos antes de explicar cómo son y en qué consisten los algoritmos de optimización global.

[33] Esto quiere decir que se deben hacer mejoras al código a partir de la gráfica cuyos nodos son bloques básicos.

3.2.1.- HERRAMIENTAS PARA OPTIMIZACIÓN.

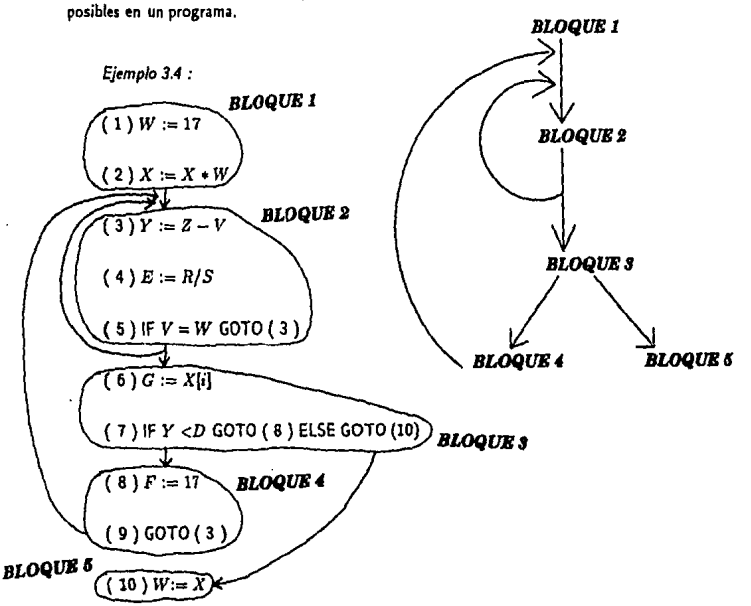
GRÁFICAS DE FLUJO.

Las gráficas de flujo son aquellas que representan todo el código de un programa y están formadas por nodos que son bloques básicos[34]. En estas gráficas se define lo siguiente:

Punto.- Es el lugar que existe entre dos instrucciones, así como el sitio antes de la primera y después de la última instrucción.

Camino.- Es una sucesión de puntos. El control de flujo se identifica con el o los caminos posibles en un programa.

Ejemplo 3.4 :



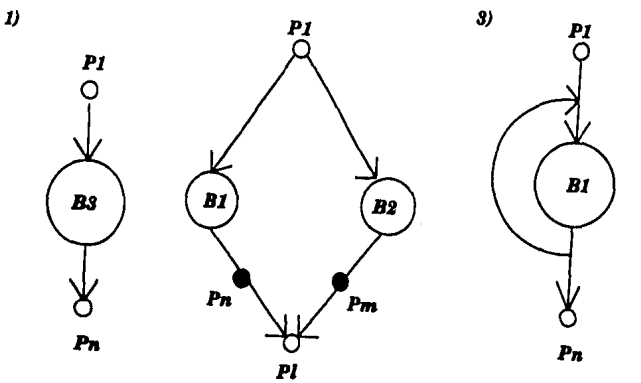
[34] En los bloques básicos es donde se hacen las optimizaciones locales.

En este código existen 11 puntos: uno antes de (1) otro entre (1) y (2), (2) y (3), (3) y (4), (4) y (5), (5) y (6), (6) y (7), (7) y (8), (8) y (9), (9) y (10) y el último al final de (10). Los caminos de este código se muestran en la gráfica.

En este ejemplo se muestra que dentro de los bloques el flujo es lineal, pero entre los bloques debido a la variedad de caminos, el flujo no necesariamente es lineal (debido a instrucciones como los *GOTO*).

Debido a la variedad de saltos que puede haber existen muchos tipos de gráficas de flujo, pero en general se forman por las siguientes estructuras básicas:

Figura 3.1



La gráfica (1) corresponde a un programa que consta de un bloque básico o que no tiene saltos en el control de flujo. La gráfica (2) corresponde a la parte de los programas donde aparecen instrucciones de la forma *IF A THEN* B_1 *ELSE* B_2, lo que provoca la aparición de bifurcaciones en la gráfica. La gráfica (3) corresponde a instrucciones equivalentes a *DO* B_1 *WHILE A* lo que significa la repetición de instrucciones de un bloque, mientras se cumpla cierta condición; este código provoca el regreso del control de flujo a bloques ya pasados y gráficas en forma de remolino o "bucles"; estas estructuras son los "lazos" definidos en el capítulo dos (sección 2.2.1).

Para cada punto de los diferentes caminos en una gráfica de flujo, existen variables que se pueden **usar** en el resto del código[35], por ejemplo, en el punto que está entre la instrucción (2) y (3) del ejemplo 3.4 se pueden usar las variables X y W debido a que ya se definieron, y en el punto que está entre la instrucción (3) y (4) ya se pueden usar también a Y, Z y V. Y así como existen variables que se pueden usar en un punto también hay expresiones que se pueden usar a partir de un punto dado.

Del hecho de que para cada camino en cada punto existen variables y expresiones que se pueden usar surgen las definiciones necesarias para la optimización global.

ANÁLISIS DE FLUJO DE DATOS.

El análisis del flujo de datos investiga cuáles variables o expresiones se pueden usar en cada punto de los caminos de las gráficas de flujo. Como dentro de los bloques básicos el control de flujo no es complicado y se supone que ya está optimizado, donde interesa aplicar este análisis es en los puntos entre un bloque y otro. Esto quiere decir que se analizará el flujo de datos de los puntos para los caminos de los bloques[36].

Como se mencionó en la sección anterior existen muchos tipos de caminos y para cada tipo existe un flujo especial de datos.

Para el análisis del flujo de datos se divide el problema en las siguientes partes:

- Alcance en las definiciones de variables.
- Expresiones disponibles.
- Variables activas.

El **alcance de las definiciones de variables** consiste en el estableciminto del tipo de valores válidos para cada variable en todo el código. En **expresiones disponibles** se determina

[35] Esto quiere decir, existen variables disponibles para ser empleadas.

[36] Con ésto se nota que la optimización global se relaciona con el flujo de datos existente entre los bloques básicos en un gráfica de flujo.

qué expresiones son válidas en cada bloque. En **variables activas** se establecerá qué variables están disponibles en cada bloque.

DEFINICIONES DE VARIABLES.

En todo código se tiene disponible un conjunto de variables que en el transcurso del programa pueden tomar diferentes valores. Cada vez que se asigna un valor a una variable se está *definiendo*. Conocer cuales definiciones está empleando cada variable en un programa es útil para la optimización (por ejemplo en la propagación de copias).

Para el análisis de las definiciones de variables en el flujo de datos se emplean las siguientes definiciones:

- En un punto dado antes de una instrucción o bloque existe un conjunto de variables definidas; *ENT[B]* = $\{A, C\}$ quiere decir que las variables activas al *ENT*rar en el bloque B son A y C.

- De igual manera, en un punto dado después de una instrucción o bloque existe un conjunto de variables definidas o redefinidas como consecuencia de asignaciones de valores a variables nuevas o usadas; *SAL[B]* = $\{D, E\}$ quiere decir que las variables activas al *SAL*ir del bloque B son D y E.

- En un bloque o instrucción existe un conjunto de variables que se *GEN*eran o activan como consecuencia de que se definen o usan; *GEN[B]* = $\{F, G\}$ quiere decir que las variables que se generan o usan en el bloque B son F y G.

- Así como se pueden definir variables también existen variables que se *DES*activan (cambian con respecto a su última definición); *DES[B]* = $\{H, I\}$ quiere decir que las variables desactivadas en el bloque B son H e I.

Ejemplo 3.5 :

Supóngase que se tiene el siguiente bloque :

Bloque 1

(1) *definición 1:* $A := B + C$

De este código se tiene que para el punto anterior al bloque, ENT [1] = $\emptyset$ (si este bloque es el principio del programa), y GEN [1] = { A } porque se está definiendo el valor de A. Y como consecuencia de lo generado en este bloque se tiene que DES [1] = { la última definición de A, anterior a esta instrucción }; y por último SAL [1] = { La definición de A como consecuencia de la instrucción (1) }, con ésto se tiene que para el punto al final del bloque está activa la definición 1.

Al emplear estas definiciones para analizar las estructuras básicas que existen en las gráficas de flujo se tiene lo siguiente (B representa a la gráfica de flujo completa):

De la figura 3.1, gráfica (1):

ENT [B] = { variables activas en el punto anterior al comienzo del bloque.}

GEN [B] = { variables generadas o redefinidas.}

DES [B] = { definiciones activas - GEN [B] }

SAL [B] = GEN [B] $\cup$ { ENT [B] - DES [B] }

De la figura 3.1, gráfica (2):

$GEN [B] = GEN [B_1] \cup GEN [B_2]$

$DES [B] = DES [B_1] \cap DES [B_2]$

$ENT [B_1] = ENT [B]$

$ENT [B_2] = ENT [B]$

$SAL [B] = SAL [B_1] \cup SAL [B_2]$

De la figura 3.1, gráfica (3):

$GEN [B] = GEN [B_1]$

$DES [B] = DES [B_1]$

$ENT [B_1] = ENT [B] \cup GEN [B_1]$

$SAL [B] = SAL [B_1]$

Para facilitar el cálculo de *DES* y *GEN* en todo código, sin importar la forma de su gráfica, basta con poder definir el conjunto de datos generados y desactivados para cada bloque porque en todo tipo de camino *DES* y *GEN* se expresan en términos de datos generados o desactivados de bloques[37].

[37] Ver figura 3.1. En las diferentes estructuras básicas *DES* y *GEN* se expresan en términos de otros datos generados o desactivados. En la gráfica 1 se emplea la definición de *DES* y *GEN*.

Para todo análisis futuro se define el conjunto de variables o expresiones activas en un punto anterior a un bloque como { *PA* }, y el conjunto de variables o expresiones activas en un punto posterior a un bloque como { *PP* }.

Basándose en la definición de *GEN* y *DES* se puede calcular el conjunto de variables generadas y redefinidas en cada punto y con esto calcular *GEN* y *DES* para cada variable en todos los bloques de un código; para hacer ésto se tiene el siguiente algoritmo donde se usa el conjunto $\{D_v\}$ que representa la última definición de la variable v (por ejemplo si en un bloque existe la instrucción $v := a + b$ y la variable v no es definida de otra forma en el bloque, entonces $\{D_v\} = \{a + b\}$), el conjunto $DES[B_i]_v$ que contiene las redefiniciones de la variable v en el bloque B_i (esto es, si a una variable previamente definida se le asigna una nueva expresión), y el conjunto $GEN[B_i]_v$ que tiene las definiciones de la variable v en el bloque B_i.

Algoritmo (3.7) Cálculo de GEN y DES para todo bloque.

Entrada: Un conjunto de bloques que forman el código generado por un compilador.

Salida: Cálculo de *GEN* y *DES* para cada bloque y cada *punto anterior (PA)* y *posterior (PP)* de un bloque[38] .

1.- Para el primer bloque (B_1), $GEN[B_1]$ = { definiciones de variables hechas por las instrucciones del código en el bloque B_1 }, y para PP_1, $GEN[B_1] = DES[B_1]$ (es decir en PA_2 se tiene que $GEN = DES$[39]).

2.- Para cada uno de los bloques B_i diferentes de B_1, GEN [B_i] = { variables definidas o usadas } y para PP_i (punto posterior del bloque B_i), GEN_{PP_i} [B_i] = $\bigcup$ {GEN [B_j]} (la unión de todos los $GEN[B_j]$ de cada bloque que alcanza dicho punto).

3.- Cada variable v generada en B_i o B_1, incluirla en el conjunto D_v.

4.- Para cada variable v en un bloque Bi ($i \neq 1$), $DES[B_i]_v = \{D_v\} - \{GEN[B_i]_v\}$

[38] El uso de *PA* o *PP* es relativo, porque los cálculos obtenidos para PP_i corresponden a los datos de PA_{i+1}.

[39] En este caso se igualan los conjuntos GEN y DES porque al principio del código no se tienen variables definidas previamente que puedan cambiar de valor y así pertenecer al conjunto DES.

5.- Para cada uno de los bloques B_i $(i \neq 1)$, $\{DES[B_i]\} = \bigcup_{i \neq 1}^{n} \{DES[B_i]_v\}$ (para cada variable v perteneciente al bloque).

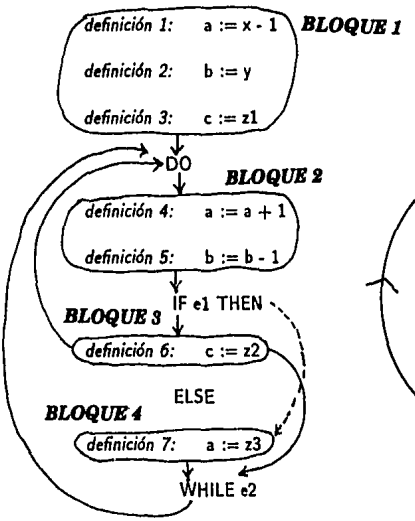
6.- Siguiendo el control de flujo, para cada punto PP_i posterior a un bloque B_i se calcula $DES[B_i] = \bigcap_{j=1}^{n} \{DES[B_j]\}$ $(1 < j < n$, para cada bloque B_j que se conecte al bloque B_i a través de un arista), y también $GEN[B_i] = \bigcup_{j=1}^{n} \{GEN[B_j]\}$, también para cada bloque que alcanza dicho punto P_i.

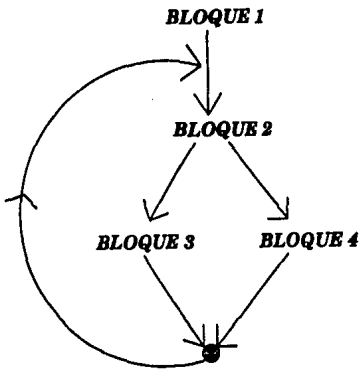
7.- Repetir el paso 6 para cada lazo.

Con los conjuntos *GEN* y *DES* se obtiene información acerca de definiciones y variables para el principio y final de cada bloque , pero con esto no se puede saber en que partes de todo el código (en todo bloque) son válidas las definiciones hechas para cada variable (es decir, el alcance de las definiciones para las variables), para esto se emplea otro algoritmo.

Ejemplo 3.6:

Calcular *GEN* y *DES* para cada bloque y punto del siguiente código:





Antes de aplicar el algoritmo para calcular GEN y DES nótese que de acuerdo a la GDA de este código existe un lazo formado por los bloques 2, 3 y 4. Y en los pasos a seguir se denota como d_i a la definición i-ésima.

Paso 1.-

Para el bloque 1, $GEN[B_1] = \{d_1, d_2, d_3\}$;

y en PP_1 se tiene que $GEN = DES$.

Paso 2.-

Para el bloque 2, $GEN_{PP_2}[B_2] = \{d_4, d_5\}$;

para el bloque 3, $GEN_{PP_3}[B_3] = \{d_6\}$;

para el bloque 4, $GEN_{PP_4}[B_4] = \{d_7\}$.

Paso 3.-

Las variables generadas en el bloque 1 son *a*, *b* y *c*, de aquí se tiene $D_a = \{x-1\}$ (d_1), $D_b = \{y\}$ (d_2), y $D_c = \{$ z1 $\}$ (d_3).

Para el bloque 2 se redefinen las variables *a* y *b*, entonces $D_a = \{d_4\} \cup \{d_1\}$

y $D_b = \{d_5\} \cup \{d_2\}$.

Para el bloque 3 se redefine *c*, así que $D_c = \{d_6\} \cup \{d_3\}$.

Y para el bloque 4 se redefine **a**, entonces $D_a = \{d_7\} \cup \{d_4, d_1\}$.

Paso 4.-

Para el bloque 2:[40] $DES[B_2]_a = \{D_a\} - \{GEN[B_2]_a\} = \{d_4, d_1\} - \{d_4\} = \{d_1\}$. $DES[B_2]_b = \{D_b\} - \{GEN[B_2]_b\} = \{d_5, d_2\} - \{d_5\} = \{d_2\}$.

Para el bloque 3: $DES[B_3]_c = \{D_c\} - \{GEN[B_3]_c\} = \{d_6, d_3\} - \{d_6\} = \{d_3\}$.

Para el bloque 4: $DES[B_4]_a = \{D_a\} - \{GEN[B_4]_a\} = \{d_7, d_4, d_1\} - \{d_7\} = \{d_4, d_1\}$.

Paso 5.-

Para el bloque 2: $DES[B_2] = DES[B_2]_a \cup DES[B_2]_b = \{d_2, d_1\}$.

Para el bloque 3: $DES[B_3] = DES[B_3]_c = \{d_3\}$.

Para el bloque 4: $DES[B_4] = DES[B_4]_a = \{d_4, d_1\}$.

Paso 6.-

$GEN_{PP_1} = GEN[B_1]$ y $DES = DES[B_1]$.

$GEN_{PP_2} = GEN[B_2]$ y $DES = DES[B_2]$.

$GEN_{PP_3} = GEN[B_3]$ y $DES = DES[B_3]$.

$GEN_{PP_4} = GEN[B_4]$ y $DES = DES[B_4]$.

Para el punto posterior a todo el código $GEN = GEN[B_3] \cup GEN[B_1] = \{d_7, d_6\}$, y $DES = DES[B_3] \cap DES[B_4] = \emptyset$.

[40] En esta parte del algoritmo todavía no se considera al ciclo existente en este código, por lo que en este punto todavía no existe d_7.

Paso 7.-

Repitiendo el paso 6 en el lazo formado por los bloques 2, 3 y 4 el único cambio es en PA_2 donde se tiene que $GEN = GEN[B_1] \cup \{d_6, d_7\}$ (el punto posterior a todo el código) $= \{d_1, d_2, d_3, d_6, d_7\}$, y $DES = \emptyset$.

De igual manera que se calculan *GEN* y *DES* basándose en las estructuras básicas para todo tipo de gráficas, también se puede hacer el cálculo de *ENT[B]* (conjunto de expresiones o variables que se pueden usar en el bloque B) y *SAL[B]* (conjunto de variables o expresiones activas al terminar el bloque B) para todo bloque.

Aunque existen analogías entre los conjuntos *SAL[B]* y *GEN[B]* la diferencia entre ellos es que *GEN[B]* es un subconjunto de *SAL[B]*.

Ejemplo 3.7 :

Sean los siguientes bloques:

Bloque 1: Definición 1.- A := B + C

Bloque 2: Definición 2.- D := F + G

En el bloque 1 $GEN[B_1] =$ { definición 1 = A }, $DES[B_1] = \emptyset$, $SAL[B_1] = [A]$, y $ENT[B_1] = \emptyset$.

Y para el bloque 2 $GEN[B_2] =$ { definición 2 = D }, $DES[B_2]) = \emptyset$, $SAL[B_2] = [A, D]$ y $ENT[B_2] = [A]$.

Apoyándose en los cálculos de *GEN* y *DES* la evaluación de *ENT* y *SAL* es más sencilla porque de acuerdo a los cálculos dados en las estructuras básicas, los conjuntos de definiciones de variables al principio y fin de cada bloque se basan en los conjuntos *GEN* y *DES*.

Algoritmo (3.8) Cálculo de ENT y SAL para todo bloque.

Entrada: Un conjunto de bloques que forman el código generado por un compilador para el que ya se han calculado *GEN [B]* y *DES [B]*, para todo bloque B.

Salida: El conjunto de variables activas antes y después de cada bloque básico (es decir, en cada PP_i y PA_i).

1.- Para todo bloque B_i, definir $\{PA_i\} = \{PP_i\} = \emptyset = ENT[B_i] = \{$ variables activas$\}$.

2.- Para el bloque 1 $PP_1 = GEN[B_1] = \{$ variables activas $\}$.

3.- Siguiendo el control de flujo, para cada bloque B_i definir:

3.1.- $\forall\, i \neq 1$ $\{PA_i\} = \bigcup\{PP_j\}$ para todo bloque j que llegue (directamente)

al bloque i por el control de flujo.

$\{PP_i\} = \{\ \{\ PA_i\} - DES[B_i]\ \} \cup GEN[B_i]$.

3.2.- $ENT[B_i] = \{\ PA_i\}$ y $SAL[B_i] = \{\ PP_i\}$

4.- Repetir el algoritmo desde el paso 3 para cada lazo que exista en el código.

Los últimos dos algoritmos dan información acerca de las definiciones de variables en un código, el algoritmo 3.7 al definir *GEN* y *DES* muestra el conjunto de variables disponibles al final de cada bloque, y el algoritmo 3.10 cuando calcula *ENT* y *SAL* para todo bloque señala el alcance de cada definición de variables a través del código[41].

[41] Es decir, para cada variable disponible muestra, en cada bloque, cuál es la última definición empleada.

Ejemplo 3.8 :

Con el código del ejemplo 3.6 al que se ha calculado *GEN* y *DES* para cada bloque calcular ENT y SAL.

Antes de aplicar el algoritmo se menciona que en el código del ejemplo 3.6 existe un lazo formado por los bloques B_2, B_3 y B_4; y que los puntos existentes son:

PA_1, PP_1, PA_2, PP_2, PA_3, PP_3, PA_4, PP_4 y el punto posterior a todo el código o punto final es PP_{final}.

Paso 1.-

$\{PA_1\} = \{PP_1\} = \{PA_2\} = \{PP_2\} = \{PA_3\} = \{PP_3\} = \{PA_4\} = \{PP_4\} = \{PA_{final}\} = \emptyset$.

Y también: $ENT[B_1] = ENT[B_2] = ENT[B_3] = ENT[B_4] =$ { variables activas} $= \emptyset$.

paso 2.-

En el bloque 1 : $GEN[B_1] = \{d_1, d_2, d_3\} = \{PP_1\}$.

paso 3.1.-

$\{PA_2\} = \{PP_3\} \cup \{PP_4\} \cup \{PP_1\} = \emptyset \cup \emptyset \cup GEN[B_1] = \{d_1, d_2, d_3\}$.

$\{PP_2\} = \{\ \{PA_2\} - DES[B_2]\ \} \cup GEN[B_2] = \{\ \{d_1, d_2, d_3\} - \{d_1, d_2\}\ \} \cup \{d_4, d_5\} = \{d_3\} \cup \{d_4, d_5\} = \{d_3, d_4, d_5\}$.

$\{PA_3\} = \{PP_2\} = \{d_3, d_4, d_5\}$.

$\{PP_3\} = \{\ \{PA_3\} - DES[B_3]\ \} \cup GEN[B_3] = \{\ \{d_3, d_4, d_5\} - \{d_3\}\ \} \cup \{d_6\} = \{d_4, d_5, d_6\}$.

$\{PA_4\} = \{PP_2\} = \{d_1, d_4, d_5\}$.

Y $\{PP_4\} = \{\ \{PA_4\} - DES[B_4]\ \} \cup GEN[B_4] = \{\ \{d_3, d_4, d_5\} - \{d_1, d_4\}\ \} \cup \{d_7\} = \{d_3, d_5, d_7\}$.

$\{PA_{final}\} = \{PP_3\} \cup \{PP_4\} = \{d_4, d_5, d_6\} \cup \{d_3, d_5, d_7\} = \{d_3, d_4, d_5, d_6, d_7\}$.

paso 3.2.-

$ENT[B_1] = \{PA_1\} = \emptyset$, $SAL[B_1] = \{PP_1\} = \{d_1, d_2, d_3\}$;

$ENT[B_2] = \{PA_2\} = \{d_1, d_2, d_3\}$, $SAL[B_2] = \{PP_2\} = \{d_3, d_4, d_5\}$;

$ENT[B_3] = \{PA_3\} = \{d_3, d_4, d_5\}$, $SAL[B_3] = \{PP_3\} = \{d_4, d_5, d_6\}$;

$ENT[B_4] = \{PA_4\} = \{d_3, d_4, d_5\}$, $SAL[B_4] = \{PP_4\} = \{d_3, d_5, d_7\}$;

SAL de todo el código es : $\{PA_{final}\} = \{d_3, d_4, d_5, d_6, d_7\}$.

paso 4.-

Repetir desde el paso 3 para los bloques 2, 3 y 4 que forman un lazo:

paso 3.1.-

$\{PA_2\} = \{PP_3\} \cup \{PP_4\} \cup \{PP_1\} = SAL[B_3] \cup SAL[B_4] \cup GEN[B_1] = \{d_4, d_5, d_6\} \cup \{d_3, d_5, d_7\} \cup \{d_1, d_2, d_3\} = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$.

$\{PP_2\} = \{\ \{\ PA_2\} - DES[B_2]\ \} \cup GEN[B_2] = \{\ \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\} - \{d_1, d_2\}\ \} \cup \{d_4, d_5\} = \{d_3, d_4, d_5, d_6, d_7\} \cup \{d_4, d_5\} = \{d_3, d_4, d_5, d_6, d_7\}$.

$\{PA_3\} = \{PP_2\} = \{d_3, d_4, d_5, d_6, d_7\}$. $\{PP_3\} = \{\ \{PA_3\} - DES[B_3]\ \} \cup GEN[B_3] = \{\ \{d_3, d_4, d_5, d_6, d_7\} - \{d_3\}\ \} \cup \{d_6\} = \{d_4, d_5, d_6, d_7\}$.

$\{PA_4\} = \{PP_2\} = \{d_3, d_4, d_5, d_6, d_7\}$. $\{PP_4\} = \{\ \{PA_4\} - DES[B_4]\ \} \cup GEN[B_4] = \{\ \{d_3, d_4, d_5, d_6, d_7\} - \{d_1, d_4\}\ \} \cup \{d_7\} = \{d_3, d_5, d_6, d_7\}$.

$\{PA_{final}\} = \{PP_3\} \cup \{PP_4\} = \{d_4, d_5, d_6, d_7\} \cup \{d_3, d_5, d_6, d_7\} = \{d_3, d_4, d_5, d_6, d_7\}$.

paso 3.2.-

$ENT[B_1] = \{PA_1\} = \emptyset$, $SAL[B_1] = \{PP_1\} = \{d_1, d_2, d_3\}$;

$ENT[B_2] = \{PA_2\} = \{d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$,

$SAL[B_2] = \{PP_2\} = \{d_3, d_4, d_5, d_6, d_7\}$;

$ENT[B_3] = \{PA_3\} = \{d_3, d_4, d_5, d_6, d_7\}$, $SAL[B_3] = \{PP_3\} = \{d_4, d_5, d_6, d_7\}$;

$ENT[B_4] = \{PA_4\} = \{d_3, d_4, d_5, d_6, d_7\}$, $SAL[B_4] = \{PP_4\} = \{d_3, d_5, d_6, d_7\}$;

SAL de todo el código es : $\{PA_{final}\} = \{d_3, d_4, d_5, d_6, d_7\}$.

Paso 4.- Como ya no hay más lazos, se termina.

Con los datos acerca de las variables generadas (GEN), redefinidas (DES) y las activas al principio y fin de cada bloque (ENT y SAL) se tienen los datos necesarios para poder aplicar criterios que mejoran el código de manera global.

EXPRESIONES DISPONIBLES.

Se dice que una expresión *(b + c)* está **disponible** en un cierto punto del código si las variables que la forman *(b, c)* no han sido redefinidas desde la última vez que se empleó dicha expresión.

Ejemplo 3.9

DEFINICIONES	EXPRESIONES DISPONIBLES
.....	Ninguna
a := b + c	
.....	sólo b + c
b := a - d	
.....	sólo a - d
c := b + c	
.....	sólo a - d
d := a - d	
.....	Ninguna

La manera de determinar las expresiones disponibles en todo bloque es similar a lo hecho en la sección anterior, sustituyendo los conjuntos manejados por los siguientes:

• Para todo bloque B existe un conjunto de expresiones disponibles al principio del bloque ($E\_ENT\ [\ B\ ]$), un conjunto de expresiones generadas o definidas ($E\_GEN\ [\ B\ ]$), otro conjunto de expresiones desactivadas ($E\_DES\ [\ B\ ]$) y un conjunto de expresiones activas al final del bloque ($E\_SAL\ [\ B\ ]$) y aplicando, en cada caso, la definición de expresión disponible.

Los conjuntos que se usan para el estudio de expresiones disponibles son análogos a los usados para determinar el alcance de las definiciones de variables, pero al tratarse de expresiones que se pueden activar o desactivar dependiendo del valor de más de una variable, el cálculo de los conjuntos es diferente. Por ejemplo, el conjunto de expresiones activas al final de un bloque depende de que cada expresión esté activa al final de cada uno de los bloques anteriores.

Para calcular los conjuntos E_GEN y E_DES se aplican métodos similares a los algoritmos usados para calcular GEN y DES en la sección anterior (algoritmos 3.8 y 3.7) sólo que en este caso se aplica a la generación de expresiones. Ya teniendo los conjuntos E_GEN y E_DES, para calcular E_ENT y E_SAL se usa el siguiente algoritmo:

Algoritmo (3.9) Cálculo de E_ENT y E_SAL para todo bloque.

Entrada: Un conjunto de bloques que forman el código generado por un compilador para el que ya se han calculado $E\_GEN\ [\ B\ ]$ y $E\_DES\ [\ B\ ]$ para todo bloque B.

Salida: Cálculo de E_ENT y E_SAL para todo bloque.

1.- $E\_ENT[B_1] = \emptyset$ y $E\_SAL[B_1] = E\text{-}GEN[B_1]$.

2.- Para todo bloque B_i $(i \neq 1)$:

2.1.- $E\_ENT[B_i] = \bigcap E\_SAL[B_j]$ para todo $\{PA_j\}$ que llegue directamente al bloque i por el control de flujo.

2.2.- $E\_SAL[B_i] = E\_GEN[B_i] \cup \{E\_ENT[B_i] - E\_DES[B_i]\}$.

VARIABLES ACTIVAS.

Para estudiar el flujo de datos es importante conocer los diferentes valores que toman las variables empleadas en el código. Una forma de hacer ésto es establecer las partes del código donde una variable mantiene un mismo valor. Este problema es similar al de calcular expresiones disponibles (algoritmo 3.9), variables activas (algoritmo 3.8) y definiciones de variables (algoritmo 3.7). Para el caso que se está tratando (variables activas) se utilizan los conjuntos *SAL_VAR[B]* (variables activas al final del bloque B), *ENT_VAR[B]* (variables activas al principio del bloque B), *VALOR[B]* (variables con un valor asignado en el bloque B) y *USO[B]* (variables que se pueden usar antes de cualquier definición en B). Estos conjuntos corresponden a *SAL, ENT, DES* y *GEN* respectivamente. Para calcular *VALOR* y *USO* se emplean el mismo algoritmo que obtiene *DES* y *GEN* (algoritmo 3.7), sólo que en este caso de las definiciones obtenidas para los conjuntos *DES* y *GEN* se toman en cuenta todas las variables; una vez hecho esto, para obtener *SAL_VAR* y *ENT_VAR* para todo bloque se sigue el algoritmo:

Algoritmo (3.10) Cálculo de ENT_VAR y SAL_VAR para todo bloque básico.

Entrada: Un conjunto de bloques que forman el código generado por un compilador para el que ya se han calculado *VALOR [B]*, *USO [B]*, *E_ENT* y *E_SAL* para todo bloque B.

Salida: Cálculo de ENT_VAR y SAL_VAR para todo bloque.

1.- $\forall B_i$ generar el conjunto $SAL\_VAR[B_j] = \bigcup E\_SAL[B_j]$ $\forall j$ que llegue al bloque i (incluyendo ciclos).

2.- Para todo bloque B generar el conjunto $ENT\_VAR[B_i] = USO[B_i] \cup \{E\_SAL[B_j] - SAL\_VAR[B_j]\}$ $\forall j \neq i$ y $\forall j$ que llegue al bloque i (incluyendo ciclos).

La diferencia de este algoritmo con los últimos dados es que el uso de la información obtenida en PP_i y PA_i es diferente (están intercambiados), y que el cálculo de la información para PP_i ya no se basa en los datos obtenidos en PA_i sino en los del final del bloque. Esto se debe a que se desea obtener el valor de cada variable al final de un bloque y esta información depende de las generaciones o desactivaciones ocurridas al final del bloque.

Con los algoritmos presentados en estas secciones se obtiene la información acerca de las definiciones de variables, su alcance, sus valores, así como el uso de expresiones. Con esta información se pueden escribir los algoritmos de optimización global.

3.2.2.- ELIMINACIÓN DE SUBEXPRESIONES COMUNES.

Con los datos proporcionados por el cálculo de variables activas y desactivas, al principio y fin de cada bloque (algoritmos 3.7 y 3.8), la aplicación del criterio para eliminar subexpresiones comunes de forma global es inmediata porque basta con revisar la información del flujo de datos en cada bloque para localizar cada expresión común y eliminarla.

Algoritmo (3.11) Eliminación de subexpresiones comunes globales

Entrada: Código representado por una gráfica de flujo.

Salida: Código sin subexpresiones comunes globales.

***1.-** Para cada expresión algebraica como $a := b + c$ definida en un bloque B_i :

***1.1.-** A partir del bloque B_i donde se encuentra la expresión o definición d_i : ($a := b+c$) buscar en el sentido del flujo del programa todas las variables definidas con la expresión $b + c$ hasta la primera redefinición de alguna de las variables b o c (es decir todas las expresiones disponibles $b + c$).

***1.2.-** Crear la variable x y sustituir la definición d_i por las instrucciones siguientes:

$$x := b + c$$

$$a := x$$

***2.-** Para cada variable v_i encontrada en 1.2 sustituir la instrucción donde se encuentra por $v_i := x$

Ejemplo 3.10

Sea el siguiente bloque:

$$a := b + c$$

$$d := e[a]$$

$$u := b + c$$

$$v := e[u]$$

Aplicando el algoritmo se obtiene lo siguiente:

$$x := b + c$$

$$a := x$$

$$d := e[a]$$

$$u := x$$

$$v := e[x]$$

Al igual que en los casos de optimización local la eliminación de subexpresiones comunes viene seguida por la eliminación de propagación de copias, porque es común la introducción de copias al eliminar subexpresiones comunes[42].

[42] Ver sección 2.1 y sección 3.1

3.2.3.- PROPAGACIÓN DE COPIAS.

En secciones anteriores se explicó que la eliminación de propagación de copias consiste en sustituir, para cada copia de la forma $a := b$, b por a en todas las demás instrucciones donde aparezca b. Para el caso de optimización local se usa esta transformación con la condición de que no se modifiquen los valores de b (ya no sería el mismo valor usado). Para el caso global se emplea la misma restricción y se expresa diciendo que se pueden eliminar las copias (de instrucciones como $a := b$) propagadas globalmente siempre y cuando en los bloques a modificar no se redefina el valor de b, además en los lazos o caminos donde el flujo del programa puede pasar más de una vez tampoco se modifica el valor de b (es decir no hay asignaciones a la variable b).

Para poder aplicar todos estos criterios y expresarlos en un algoritmo es necesario tener la información de las definiciones, uso de variables, y expresiones disponibles de todos los bloques donde se encuentran las copias y si dichas copias de variables asignan el mismo valor o no.[43] El algoritmo es el siguiente:

Algoritmo (3.12) Eliminación de propagación de copias.

Entrada: Gráfica de flujo con información de expresiones disponibles, definiciones y uso de variables (E_GEN, E_DES, E_ENT, E_SAL, USO, GEN y DES).

Salida: Gráfica de flujo sin copias propagadas.

***1.-** Para cada instrucción de copia $a := b$

***1.1.-** Los usos de a que tienen el valor de b pueden encontrarse por medio de los conjuntos ENT_VAR y SAL_VAR (algoritmo 3.10)[44].

***2.-** Si el uso encontrado se halla en un bloque B, tal que se encuentra en el conjunto E_ENT[B] y no ha habido redefinición de a ó b, entonces sustituir la definición por b.

---

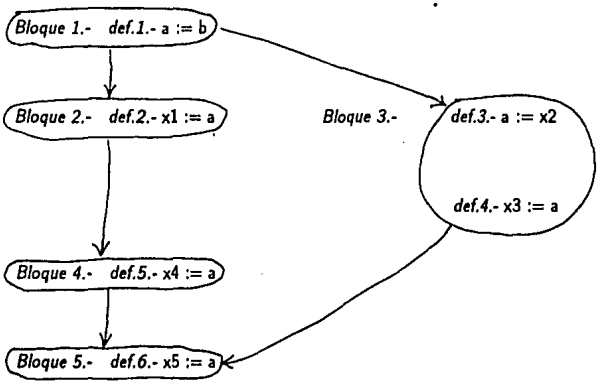
[43] Como las expresiones que interesa conocer son las de copias, se utiliza el algoritmo 3.9 que calcula el uso de expresiones disponibles

[44] Para cada uso de a que hace el valor de b.

Ejemplo 3.11

Considérese un código con 5 bloques (por simplicidad sólo se indican las instrucciones relevantes para este algoritmo).

Sea el siguiente código y su gráfica:



La información para cada bloque es la siguiente:

Bloque 1.-

$E\_GEN[B_1] = \{ a := b \}$; $E\_DES[B_1] = \emptyset$; $GEN[B_1] = (\text{def. } 1)\ d_1$;

$DES[B_1] = \emptyset$; $E\_ENT[B_1] = \emptyset$; $E\_SAL[B_1] = \{ a := b \}$; $USO[B_1] = \{ a, b \}$.

Bloque 2.-

$E\_GEN[B_2] = \{x_1 := a \}$; $E\_DES[B_2] = \emptyset$; $GEN[B_2] = d_2$; $DES[B_2] = \emptyset$;
$E\_ENT[B_2] = \{ a := b \}$; $E\_SAL[B_2] = \{ a := b \} \cup \{ x1 := a \}$; $USO[B_2] = a, b, x1$.

Bloque 3.-

E_GEN[B_3] = { a := x2; x3 := a } ; E_DES[B_3] = { a := b }; GEN[B_3] = $\{d_3, d_4\}$; DES[B_3] = $\{d_1\}$; E_ENT[B_3] = { a := b }; E_SAL[B_3] = { a := x2; x3 := a }; USO[B_3] = *a, b, x2, x3.*

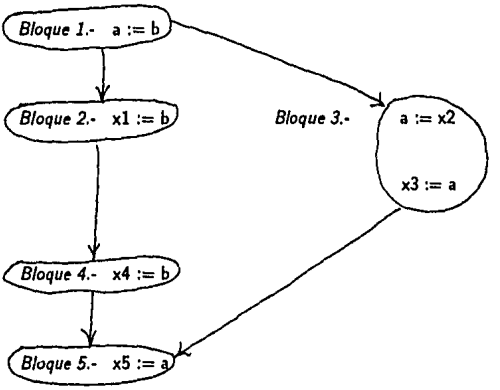
Bloque 4.-

E_GEN[B_4] = { x4 := a }; E_DES[B_4] = $\emptyset$; GEN[B_4] = $\{d_5\}$; DES[B_4] = $\emptyset$; E_ENT[B_4] = { a := b; x1 := a }; E_SAL[B_4] = { a := b; x1 := a; } $\cup$ { x4 := a }; USO[B_4] = *a, b, x1, x4.*

Bloque 5.-

E_GEN[B_5] = { x5 := a }; E_DES[B_5] = $\emptyset$; GEN[B_5] = $\{d_6\}$; DES[B_5] = { a:=b; x1:=a; x4:=a; a:=x2; x3:=a }; E_ENT[B_5] = $\emptyset$; E_SAL[B_5] = { x5 := a }; USO[B_5] = *a, b, x1, x2, x3, x4, x5.*

Basándose en estos datos y aplicando el algoritmo, este código queda así:



3.2.4.- OPTIMIZACIÓN EN LAZOS.

Para las transformaciones hechas en las secciones anteriores se aplicaron los criterios que mejoran el código de toda una gráfica de flujo con información de variables y expresiones. Esta manera de mejorar el código no considera el hecho de que, en los lazos de las gráficas, se puede mejorar el tiempo de ejecución si se localizan las variables o expresiones que permanecen invariantes en el lazo. Una variable a es invariante en un lazo si toda modificación de dicha variable se encuentra fuera del lazo. De igual manera, toda expresión utilizada es invariante si sus miembros no son modificados en el lazo, toda asignación de una expresión invariante también es invariante. La unión de variables y expresiones invariantes forman todo el conjunto de instrucciones que *posiblemente* se puede trasladar fuera del lazo.

Para detectar los posibles elementos invariantes se necesita información en cada bloque acerca de las definiciones de variables y de la disposición de expresiones.

Con los conceptos de código invariante se da el siguiente algoritmo para detectar el código invariante en un lazo.

Algoritmo (3.13) Detección de código invariante en lazos.

Entrada: Una gráfica de flujo con información de definiciones de variables y disposición de expresiones.

Salida: Detección de código invariante en todos los lazos del código.

Para cada lazo en el código.

Marcar como invariante toda expresión de la forma $a := b + c$ si las variables que la integran están definidas fuera del lazo.

Este algoritmo se refiere al caso de variables o expresiones definidas nada más en un lazo y que fuera de ese ciclo nunca más se vuelven a utilizar o definir.

Ejemplo 3.12

Sea el siguiente código:

Bloque 1

(1.1) a := 17

(1.2) b := x

(1.3) c := y

IF c <77 GOTO (3.1) ELSE GOTO (2.1)

Bloque 2

(2.1) u := a + b

(2.2) d := 7

(2.3) i := i + 1

GOTO (5)

Bloque 3

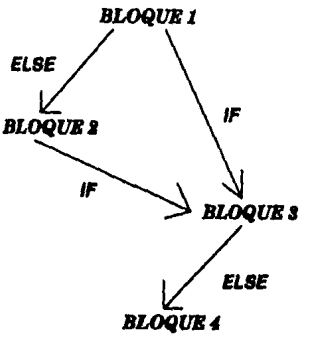
(3.1) v := a + b

(3.2) i := i + 1

IF u <77 GOTO (1) ELSE GOTO (4.1)

(4.1)

(Resto del programa)



Supóngase que la variable i definida en el lazo del bloque 2 no se utiliza en otro lugar, pero la variable d si es usada, entonces al aplicar el algoritmo para descubrir el código invariante en el lazo que aparece en esta gráfica, se tiene que las instrucciones invariantes son $u := a + b$, $v := a + b$.

Para optimizar los lazos se localiza el conjunto de instrucciones innecesarias en el ciclo y se traslada fuera del lazo. Detectando el código invariante se obtiene el conjunto al que pertenecen las instrucciones que se pueden sacar del lazo. Pero no es posible sacar toda instrucción invariante del lazo, en el ejemplo anterior $i := i + 1$ es necesaria dentro del lazo.

Para poder trasladar código fuera de un lazo es necesario asegurase que este cambio no afecta el funcionamiento del resto del código porque aunque una instrucción sea innecesaria en un lado puede modificar sensiblemente el programa en otro lado. Toda instrucción invariante (de la forma a:=*invariante*) debe cumplir las siguientes condiciones para poder trasladarla:

- 1.- No debe haber otra asignación a la variable a en el lazo.

- 2.- Toda instrucción que utilice a la variable a debe manejar la definición hecha antes del lazo.

- 3.- El bloque donde se encuentra la definición de la invariante domina a todos los nodos de salida del lazo, es decir, todo camino del flujo que salga del lazo debe pasar por el bloque donde está la definición de la invariante.

Ejemplo 3.13

Bloque 1

(1.1) i := 1

Bloque 2

(2.1) i := 3

(2.2) IF a <b GOTO (3.1) ELSE GOTO (4.1)

Bloque 3

(3.1) i := 2

(3.2) a := a + 1

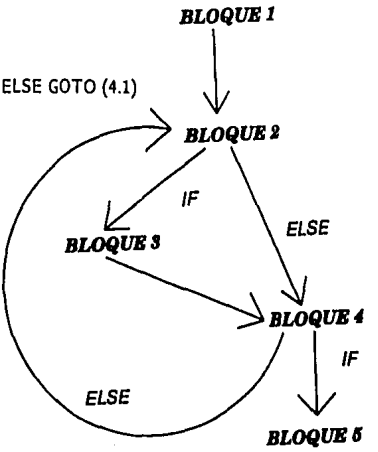
Bloque 4

(4.1) b := b - 1

(4.2) IF b <20 GOTO (5.1) ELSE GOTO (2.1)

Bloque 5

(5.1) j := i



De acuerdo al algoritmo para detectar código invariante la instrucción *i := 2* en el bloque 3 es código invariante en el lazo, pero esta instrucción no se puede trasladar antes del lazo

porque se puede cambiar el valor asignado a *j* en el bloque 5, para los casos donde el flujo del programa no pasa por el bloque 3. Esto ocurrió porque el bloque 3 donde se encuentra *i:=2* no domina al bloque 4, que es la única salida del lazo, esto es, para salir del lazo existe un camino que no pasa por el bloque 3.

Si se traslada $i := 3$ fuera del lazo formado por B_2, B_3, B_4, al ejecutarse el código la variable *i* siempre valdrá 2 si se recorre sólo una vez el bloque 3, esto es, para el camino B_2, B_3, B_4, B_2, B_4, B_5, la variable *i* tiene un valor diferente en código original y en el código donde se traslada la instrucción. Esto se debe a que dentro del lazo hubo más de una asignación a la variable *i* que forma parte de una instrucción invariante.

Teniendo en cuenta el algoritmo para encontrar invariantes en lazos y los criterios para trasladar código se obtiene el algoritmo para optimizar lazos.

Algoritmo (3.14) Traslado de código en lazos.

Entrada: Gráfica de flujo con información de definiciones de variables y disposición de expresiones.

Salida: Gráfica con lazos optimizados, es decir donde se ha trasladado código invariante en los casos posibles.

***1.-** Para cada lazo, detectar el código invariante (algoritmo 3.13).

***2.-** Para cada elemento invariante:

Si

- El elemento se encuentra en un bloque que domine todas las salidas del lazo .
- La variable a la que se asigna la expresión invariante se define en otro lugar fuera del lazo.

- No hay otra asignación en el lazo a la variable.

Entonces, trasladar dicho código al punto anterior al comienzo del lazo. Todos los traslados se hacen de acuerdo al orden en que se obtienen.

OPTIMIZACIÓN GLOBAL (RESÚMEN)

Todos los algoritmos para optimizar globalmente el código se aplican, generalmente, en el orden en que se dieron (porque el código inactivo se produce después de la eliminación de subexpresiones comunes, etc.) y es recomendable optimizar el código en la gráfica de flujo antes de mejorar los lazos porque de lo contrario esta mejora se puede complicar.

Análogamente, primero debe hacerse la optimización local y después la global.

CONCLUSIONES

En este trabajo no se pudieron desarrollar todos los métodos y técnicas para optimizar código; debido a que para cada tipo de código se pueden hacer reducciones en tamaño y tiempo de ejecución utilizando recursos propios del lenguaje fuente y lenguaje objeto, esto es, cada lenguaje tiene instrucciones que pueden ahorrar código o aumentar la velocidad de ejecución, que no necesariamente son comunes a todo lenguaje.

Para cada *software* y *hardware* existen optimizaciones que les son propias y enumerarlas todas no tendría sentido, el mérito de este trabajo es que se mostraron los métodos que en general debe seguir cada optimización sin importar la manera de implantarse en una máquina.

Aunque no se escribieron *todos las optimizaciones* posibles, se analizaron los métodos que deben seguirse para obtener un código óptimo para todo tipo de compilador en cualquier lenguaje y esto fue posible debido a que se manejaron algoritmos no escritos en un lenguaje de programación particular.

Los algoritmos expuestos se basan en la información obtenida en la bibliografía. La aportación de este trabajo fue la creación de algoritmos intermedios necesarios para la comprensión de algoritmos importantes, también se formularon algoritmos para temas que la bibliografía sólo mencionaba como criterios de optimización, además la mayoría de los ejemplos se formularon con base en experiencias del autor. Debido a la relación del tema con la teoría de algoritmos se tuvo que hacer una exposición acerca de la teoría en que se fundamenta la construcción de computadoras y compiladores. Para el desarrollo de la teoría de compiladores este trabajo se basó en la experiencia obtenida en la construcción de un compilador de *pascal* utilizando *lex* y *yacc*.

Debido a lo general del tema desarrollado se logró el propósito de establecer las principales bases para la construcción de compiladores que generen código optimizado. Pero por la misma razón de lo general que es el tema de la optimización de código, se podría extender este trabajo al buscar algoritmos para diferentes tipos de arquitecturas de computadoras.

BIBLIOGRAFIA.

Lee Goldschlager [1986]; *Introducción Moderna a la ciencia de la computación*; Prentice Hall.

Aho, Alfred [1977]; *Compiladores*; Addison-Wesley

Pollack, Barry [1980]; *Compiladores.*

F.R.A Hopgood [1992]; *Compiling techniques*; Addison-Wesley.

Bauer [1985]; *Compiler construction*; Spriger-Verlag.

Allen, F.E. Cocke J [1988]; *A catalogue of optimizing transformations.*

Lee, jhon, A [1990]; *The anatomy of a compiler*;

Levine, Guillermo [1990]; *Introducción a la computación y a la programación estructurada*; McGraw-Hill.

Hopcroft, Ullman [1977]; *Compiladores*; Addison-Wesley.

INDICE ANALITICO.

A

E

Ejemplo

Ejemplo

H

I

L

M

R

S

T

V