ANÁLISIS SEMÁNTICO

José Sánchez Juárez

Septiembre, 2018

II

Índice general

Índice de figuras

Índice de cuadros

Análisis semántico y generación de código intermedio

> La mente intuitiva es un regalo sagrado y la mente racional una sirviente fiel. Hemos creado una sociedad que honra a los sirvientes y que ha olvidado los regalos.
>
> Albert Einstein.
> 1879−1955

La etapa de análisis semántico de un compilador es la que acopla las definiciones de las variables con su presencia en el programa, verifica que cada expresión tenga un tipo correcto y traduce la sintaxis abstracta en una representación más simple, adecuada para generar código máquina [**?**].

Podemos decirlo de otra manera: como asignar un significado a cada producción. Se hace uso de las asociaciones entre las producciones y sus significados que proporcionan el primer paso para la traducción del programa. Porque la secuencia de producciones guían la generación de código intermedio, al que llamamos traducción dirigida por la sintaxis [1].

Atributos

Un formalismo que ha sido propuesto para desempeñar análisis sensible al contexto es el atributo de las gramáticas, o gramáticas libre de contexto atribuidas. Un atributo de gramática consiste de una gramática libre de contexto aumentada por un conjunto de reglas que especifican el cálculo.

DEFINICIÓN 1 (Atributo.) *Es un valor acoplado a uno o más nodos del árbol de análisis sintáctico [5].*

Cada regla define un valor, o atributo, en términos de los valores de otros atributos. Las reglas asocian el atributo con un símbolo especifico de la gramática: cada instancia del símbolo de la gramática que ocurre en un árbol de análisis

sintáctico tiene una instancia correspondiente del atributo. Las reglas son funcionales; ellas no implican un orden de evaluación especifico y solamente definen cada valor del atributo.

Atributos sintetizados

DEFINICIÓN 2 (Atributo sintetizado.) *Es un atributo definido totalmente en términos de atributos del nodo, sus hijos y constantes [5].*

Ejemplo de la aplicación de atributos sintetizados

Como ejemplo preliminar de la asociación entre producciones y significados. Consideremos el análisis sintáctico ascendente. Si adjuntamos una interpretación apropiada para cada una de las producciones, tenemos:

PRODUCCIÓN	ACCIÓN SEMÁNTICA
$L \to En$	$\{print(E.val)\}$
$E \to E_1 + T$	$\{E.val := E_1.val + T.val\}$
$E \to T$	$\{E.val := T.val\}$
$T \to T_1 * F$	$\{T.val := T_1.val * F.val\}$
$T \to F$	$\{T.val := F.val\}$
$F \to (E)$	$\{F.val := E.val\}$
$F \to i$	$\{F.val := i.lexema\}$

Cuadro 1: Tabla de acciones semánticas para la gramática G_1.

Por ejemplo, la interpretación de $E \to E_1 + T$ nos dice que para obtener E, debemos agregar los valores que E_1 y T evaluaron, como lo expresa la acción semántica correspondiente del cuadro 1 . Lo mismo se hace para la producción $T \to T_1 * F$.

Supongamos que repetimos el análisis ascendente para la expresión $3*5+4n$, y cada vez que reducimos por una de las producciones del cuadro 1 , generamos código para llevar a cabo el cálculo correspondiente, utilizando las interpretaciones enumeradas en el cuadro 1 . A fin de distinguir los identificadores, supongamos que los lexemas son 3, 5 y 4.

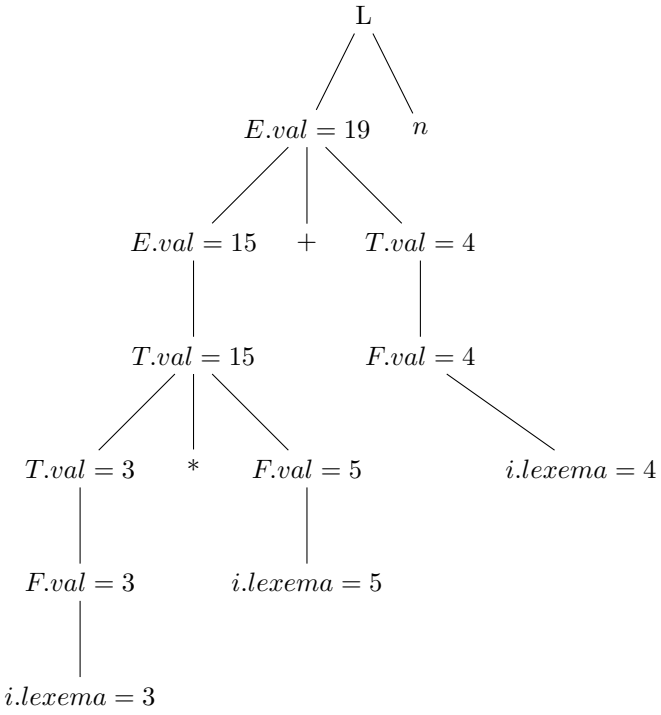


Figura 1: Árbol de análisis sintáctico adornado de la gramática G_1.

Del árbol de la figura 1 las hojas representan la expresión $3 * 5 + 4n$, el cual se va generando de acuerdo al cuadro 2 , usando las acciones semánticas que se expresan en el cuadro 1 . El árbol sintáctico de la figura 1 indica la manera como se va haciendo la valoración de cada uno de los símbolos de la gramática. Donde los valores obtenidos son los atributos que son del tipo de **atributos sintetizados**.

PILA	ENTRADA	PRODUCCIÓN	ACCIÓN SEMÁNTICA
$\$$	$\uparrow i * i + i$	$F \to i$	$\{i.lexema = 3\}$
$\$i$	$i \uparrow *i + i$	$F \to i$	$\{F.val = i.lexema\}$
$\$F$	$i \uparrow *i + i$	$F \to T$	$\{E.val := T.val\}$
$\$T$	$i \uparrow *i + i$	$T \to F$	$\{T.val := F.val\}$
$\$E$	$i \uparrow *i + i$	$E \to T$	$\{E.val := T.val\}$
$\$E*$	$i* \uparrow i + i$	$F \to i$	$\{F.val := E.val\}$
$\$E * i$	$i* \uparrow i + i$	$F \to i$	$\{i.lexema := 5\}$
$\$E * F$	$i* \uparrow i + i$	$F \to i$	$\{T.val := F.val\}$
$\$E * F$	$i* \uparrow i + i$	$T \to F$	$\{T.val := F.val\}$
$\$E * T$	$i* \uparrow i + i$	$E \to T$	$\{E.val := T.val\}$
$\$E * E$	$i* \uparrow i + i$	$E \to T$	$\{E.val := T.val\}$
$\$E * E$	$i* \uparrow i + i$	$E \to E * T$	$\{E.val := E.val * T.val\}$
$\$E$	$i * i \uparrow +i$	$E \to E * T$	$\{E.val := E.val * T.val\}$
$\$E+$	$i * i+ \uparrow i$	$F \to i$	$\{E.val := E.val * T.val\}$
$\$E + i$	$i * i + i \uparrow$	$F \to i$	$\{E.val := E.val * T.val\}$
$\$E + T$	$i * i + i \uparrow$	$T \to F$	$\{E.val := E.val * T.val\}$
$\$E + T$	$i * i + i \uparrow$	$E \to E + T$	$\{E.val := E.val + T.val\}$
$\$E$	$i * i + i \uparrow$	$E \to E + T$	$\{E.val := E.val + T.val\}$
$\$E$	$i * i + i \uparrow$	$E \to E + T$	$\{print(E.val)\}$

Cuadro 2: Tabla que muestra el uso de una pila para hacer la traducción de la cadena $3 * 5 + 4n$ de la gramática G_1.

Atributos heredados

DEFINICIÓN 3 (Atributo heredado.) *Es un atributo definido totalmente en términos de sus propios atributos del nodo y los de sus hermanos o su padre en el árbol de análisis sintáctico (más constantes) [5].*

Ejemplo de la aplicación de atributos heredados.

Se presenta la gramática con sus acciones semánticas en el cuadro 3 .

PRODUCCIÓN	ACCIÓN SEMÁNTICA
$D \to TL$	$\{L.her := T.tipo\}$
$T \to int$	$\{T.tipo := integer\}$
$T \to real$	$\{T.tipo := real\}$
$L \to L_1, id$	$\{L_1.her := L.her \quad funcionAñade(id.entrada, L.her)\}$
$L \to id$	$\{funcionAñade(id.entrada, L.her)\}$

Cuadro 3: Tabla de acciones semánticas para la gramática G_2.

Se analiza para la cadena $real \quad id_1, id_2, id_3$.

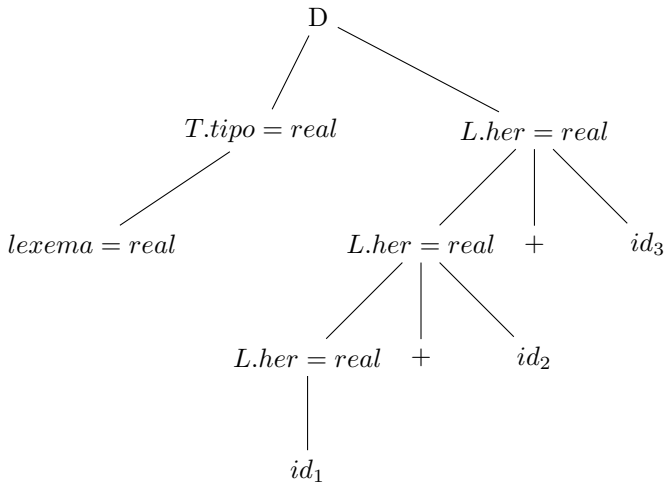


Figura 2: Árbol de análisis sintáctico adornado de la gramática G_2.

Del árbol de la figura 2 las hojas representan la expresión $real \;\; id_1, id_2, id_3$, el cual se va generando de acuerdo al cuadro 4 , usando las acciones semánticas que se expresan en el cuadro 3 . El árbol sintáctico de la figura 2 indica la manera como se va haciendo la valoración de cada uno de los símbolos de la gramática. Donde los valores obtenidos son los atributos que son del tipo de **atributos heredados**.

PILA	ENTRADA	PRODUCCIÓN	ACCIÓN SEMÁNTICA
\$	$\uparrow real \;\; id_1, id_2, id_3$	$D \to TL$	$\{lexema = real\}$
\$$real$	$real \uparrow id_1, id_2, id_3$	$T \to real$	$\{T.tipo = lexema\}$
\$$Tid$	$real \;\; id_1 \uparrow, id_2, id_3$	$L \to id$	$\{L.her := lexema\}$
\$$TL$	$real \;\; id_1 \uparrow, id_2, id_3$	$L \to L, id$	$\{lexema := id_1\}$
\$$TL,$	$real \;\; id_1, \uparrow id_2, id_3$	$L \to L, id$	$\{L.her := real\}$
\$$TL, id$	$real \;\; id_1, id_2 \uparrow, id_3$	$L \to L, id$	$\{F.val := E.val\}$
\$$TL$	$real \;\; id_1, id_2, \uparrow id_3$	$L \to L, id$	$\{i.lexema := 5\}$
\$$TL,$	$real \;\; id_1, id_2, \uparrow id_3$	$F \to i$	$\{T.val := F.val\}$
\$$TL, id$	$real \;\; id_1, id_2, id_3 \uparrow$	$L \to L, id$	$\{T.val := F.val\}$
\$$TL$	$real \;\; id_1, id_2, id_3 \uparrow$	$L \to id$	$\{E.val := T.val\}$
\$$D$	$real \;\; id_1, id_2, id_3 \uparrow$	$D \to TL$	$\{E.val := T.val\}$

Cuadro 4: Tabla que muestra el uso de una pila para hacer la traducción de la cadena $real \;\; id_1, id_2, id_3$ de la gramática G_2.

Diseño de tablas de símbolos para soportar llamadas a funciones y/o procedimientos y ámbito de variables

Un concepto importante en lenguajes de programación es la capacidad de nombrar objetos como variables, funciones y tipos. **Cada uno de esos objetos nombrados tendrán una declaración, donde el nombre se define como un sinónimo del objeto. Esto es llamado enlace**. Cada nombre también tendrá una serie de usos, donde el nombre es utilizado como referencia al objeto al que está vinculado. A menudo, la declaración de un nombre tiene un alcance limitado: una parte del programa donde el nombre será visible. Tales declaraciones se llaman declaraciones locales, mientras que una declaración que hace que el nombre declarado sea visible en todo el programa se llama declaración global. Puede suceder que el mismo nombre sea declarado en varios ámbitos anidados. En este caso, es normal que la declaración más cercana a un uso del nombre será la que defina el uso particular. El contexto más cercano está relacionado con el árbol de sintaxis del programa: el alcance de una declaración será un subárbol del árbol de sintaxis y las declaraciones anidadas darán lugar a ámbitos que son subárboles anidados. La declaración más cercana de un nombre es por lo tanto la declaración correspondiente al subárbol más pequeño que encierra el uso del nombre. El alcance basado de esta manera en la estructura del árbol de sintaxis se llama enlace estático o léxico y es la regla de alcance más común en los programas modernos lenguajes de gramática. Algunos idiomas tienen dinámica vinculante, cuando la declaración que se encontró más recientemente durante la ejecución del programa define el uso actual del nombre. Por su naturaleza, el enlace dinámico no se puede resolver en tiempo de compilación.

Un compilador necesitará hacer un seguimiento de los nombres y los objetos que son obligados a, que cualquier uso de un nombre se atribuirá correctamente a su declaración. Esto se hace típicamente usando una tabla de símbolos (o entorno, como es llamado a veces) [4].

Implementación de la tabla

Hay muchas formas de implementar las tablas de símbolos, pero la más importante es la que distingue cómo se manejan los alcances. Esto se puede hacer usando una estructura de datos persistente (o funcional), o puede hacerse usando una estructura de datos imperativa (o actualizada de forma destructiva). Una estructura de datos persistente tiene la propiedad de que ninguna operación en la estructura lo destruirá. Conceptualmente, se hace una nueva copia de la estructura de datos cada vez que una operación la actualice, preservando así el cambio de la estructura anterior. Esto significa que es trivial restablecer la vieja tabla de símbolos cuando se sale de un alcance, ya que ha sido preservada por la naturaleza persistente de la estructura de datos. En la práctica, solo se copia una pequeña porción de la estructura de datos, la mayoría se comparte con la versión anterior. En el enfoque imperativo, solo existe una copia de la tabla de símbolos, por lo que se requieren acciones explícitas para almacenar la

información necesaria para restaurar la tabla de símbolos a un estado anterior. Esto se puede hacer usando una pila. Cuando una actualización se hace, el enlace anterior de un nombre que se sobrescribe se registra (se hace un push) en la pila. Cuando se ingresa un nuevo alcance, se inserta un marcador en la pila. Cuando se sale del alcance, se usan los enlaces en la pila (hasta el marcador) para restablecer la vieja tabla de símbolos. Las ligaduras y el marcador se deshacen fuera de la pila en el proceso, devolviendo la pila al estado en que estaba antes del alcance del que fue ingresado [4].

Para implementar la tabla de símbolos se requieren las siguientes operaciones:

1. Crear una tabla de símbolos vacía (**crear_tablaSimbolos**).
2. Registrar la apertura de un bloque nuevo (**meter_bloque**).
3. Reiniciar la tabla de símbolos al estado anterior del último bloque (**sacar_bloque**).
4. Agregar la entrada de identificadores id a la tabla de símbolos. Esto contiene el apuntador a la declaración pasada en el (**meter_identificador(id, apt declaración)**).
5. Buscar la definición presente del id y regresar el apuntador de la declaración, si es que existe (**buscar_id(id)**).

Y se proponen los siguientes procedimientos y la función para construir las operaciones que se enlistaron arriba:

Listing 1: Procedimiento para crear la tabla de símbolos

```
1 procedimiento crear_tablaSimbolos;
2     begin
3         crear pila vacia para meter bloques
4     end;
```

Listing 2: Procedimiento para meter bloques

```
1 procedimiento meter_bloques;
2     begin
3         push la entrada nueva para el bloque nuevo
4     end;
```

Listing 3: Procedimiento para sacar bloques

```
1 procedimiento sacar_bloques;
2     begin
3         FOR para cada declaracion de entrada para el
4         bloque presente DO
5             borrar entrada
6         OD;
7         pop el bloque de entrada de la pila
8     end;
```

Listing 4: Procedimiento para meter identificadores

```
procedimiento meter_id(id:idno; apt nodo);
    begin
        IF si una entrada id ya existe para el bloque
        THEN error("Declaracion repetida")
        FI;
        crear entrada nueva con declaracion y numero
        de bloque

        agregar esta entrada al final de la lista lineal
        para el id

        agregar esta entrada al final de la lista lineal
        para el bloque
    end;
```

Listing 5: Función buscar identificador

```
funcion buscar_id(id: idno) apt nodo;
    begin
        IF la lista para id esta vacia
        THEN error("Identificador no declarado")
        ELSE regresa(Valor de la declaracion de la
        primera entrada en la lista para id)
        FI
    end
```

Funciones y procedimientos

Un cuerpo de procedimiento único usa (en la mayoría de los lenguajes) un número finito de variables. Se puede mapear estas variables en un (posiblemente más pequeño) conjunto de registros. Un programa que usa procedimientos recursivos o funciones, pueden usar un número ilimitado de variables, ya que cada invocación de la función tiene su propio conjunto de variables, y no hay límite en la profundidad de la recursión. No podemos esperar mantener todas estas variables en registros, entonces usaremos memoria para algunos de estos. La idea básica es que solo varían las funciones locales de la función activa (llamada más recientemente) se mantendrán en registros. Todas las demás variables se mantendrán en la memoria. Cuando se llama a una función, todas las variables activas de la función llamada (la persona que llama) se almacenará en la memoria para que los registros se liberen para ser utilizado por la función llamada (el llamado).

Cuando el destinatario regresa, las variables almacenadas se vuelven a cargar en los registros. Es conveniente usar un apilamiento para este almacenamiento y cargar, empujando los contenidos del registro en la pila cuando se deben guardar y volver a colocarlos en los registros cuando deben estar restaurados como una pila es (en principio) ilimitada, esto encaja bien con la idea de recursividad ilimitada. La pila también se puede usar para otros fines [4]:

1. Se puede reservar espacio en la pila para variables que deben ser desplazadas a la memoria. Se usa una dirección constante (dirección x) para desplazar una variable x. Cuando se usa una pila, la dirección x es en realidad una compensación relativa a un puntero de pila. Esto hace que el código de desplazamiento sea un poco más complejo, pero tiene la ventaja de que los registros desplazados ya están guardados en la pila cuando se llame a la función, por lo que no necesitan almacenarse nuevamente.

2. Los parámetros para llamadas de función se pueden pasar a la pila, es decir, escritos en la parte superior de la pila por la persona que llama y que leyó del destinatario.

3. La dirección de la instrucción donde la ejecución debe reanudarse después de la llamada de retorno (la dirección de retorno) se puede almacenar en la pila.

4. Se decide mantener solo las variables locales en los registros, variables que están dentro del alcance de una función pero no están declaradas localmente en la función deben residir en la memoria. Es conveniente acceder a ellos a través de la pila.

5. Las matrices y los registros asignados localmente en una función se pueden asignar en la pila.

Implementación de la tabla de procedimientos

Frente al código generado para un cuerpo de la función, se necesita escribir un código que lea los registros de los parámetros del registro de activación. Este código se llama el prólogo de la función. Del mismo modo, después de la función, se necesita escribir código para almacenar el valor de retorno calculado en la activación y que salte a la dirección de retorno que se almacenó al llamar el registro de activación. Para el diseño del registro de activación que se muestra en la figura 10.1, el prólogo y el epílogo adecuados se muestran en la figura 10.2. Se debe tomar en cuenta que, aunque se ha utilizado una notación similar al lenguaje intermedio, se ha ampliado esto un poco: utilizando M [] y GOTO como expresiones en general con argumentos [4].

Variables

Hasta ahora se ha supuesto que todas las variables utilizadas en una función son locales a esa función, pero la mayoría de los lenguajes de alto nivel también permiten el acceso a las funciones de variables que no están declaradas localmente en las funciones mismas.

El código en C solo admite declaraciones de variables a nivel global. Todas las variables deben ser declaradas en los segmentos de datos o bss (bss significa "block staning with symbol".), según se hayan inicializado o no. Hay cuatro clases de almacenamiento disponibles (representadas en lenguaje C, en el cuadro 5):

	static	no static	
subrutina (en el segmento del texto)	private	public	external
variable inutilizada (en el segmento del bss)	private	common	
variable inicializada (en el segmento de datos)	private	public	

Cuadro 5: Tabla de variables.

private Se asigna espacio para la variable, pero no se puede acceder a la variable desde fuera del archivo actual. En C, esta clase se usa para todas las variables estáticas, sean ellas locales o globales. Las variables inicializadas entran en el segmento de datos, otras variables entran en el segmento bss.

public Se asigna espacio para la variable, y se puede acceder a la variable desde cualquier archivo en el programa actual. En C, esta clase se usa para todos los inicializados variables globales no estáticas Es ilegal para dos variables públicas en el mismo programa para tener el mismo nombre, incluso si están declarados en diferentes archivos. Dado que las variables públicas deben inicializarse cuando se declaran, deben estar en el segmento de datos.

common El espacio para esta variable es asignado por el vinculador. Si una variable con un dado nombre es declarado común en un módulo y público en otro, luego el la definición pública tiene prioridad. Si no hay nada más que común definiciones para una variable, luego el enlazador asigna espacio para esa variable en el segmento bss. C usa esta clase de almacenamiento para todas las variables globales sin inicializarlas.

external El espacio para esta variable se asigna en otro lugar. Si una etiqueta es externa, una etiqueta idéntica debe ser declarada common o public en algún otro módulo del programa. Esta clase de almacenamiento no se usa para las variables en la aplicación corriente, todas las cuales son common, public o private. Sin embargo, se utilizan para subrutinas [**?**].

Implementación de la tabla de Variables

Se usarán las funciones hash para la capa de la base de datos de la tabla de símbolos. El sistema tabla-hash utiliza dos estructuras de datos: la tabla hash en sí misma es una matriz de punteros a "cubos", cada uno de los cubos es un único registro de base de datos. Los cubos están organizados como una lista enlazada. Cuando el hash de dos claves con el mismo valor, el nodo en conflicto se inserta al comienzo de la lista. Esta lista de cubos está doblemente enlazada: cada cubeta contiene un puntero a ambos, al predecesor y sucesor en la lista. Los nodos arbitrarios se pueden eliminar de la mitad dle de una cadena sin tener que atravesar toda la cadena. Debe revisar estas funciones ahora. Las funciones hash básicas están bien para aplicaciones simples de tablas de símbolos. Ellos fueron usados para buenos efectos en occs y LLama, por ejemplo. La mayoría de los lenguajes requieren un poco más de complejidad, sin embargo. En primer lugar, muchas operaciones internas requieren el compilador para tratar todas las variables declaradas en un nivel de anidación común como un solo bloque. Por

ejemplo, las variables locales en C se pueden declarar al comienzo de cualquier llave con llave declaración compuesta delimitada. El cuerpo de una función no es un caso especial, se trata de identidades de una declaración compuesta adjunta a una declaración while, por ejemplo. El alcance de cualquier variable local se define por los límites de la declaración compuesta. Para todos los practicantes para fines prácticos, una variable deja de existir cuando el corsé se ha cerrado para terminar ese bloque en el cual es declarado ese proceso. La variable debe eliminarse de la tabla de símbolos en ese tiempo. (No se confunda la compilación y el tiempo de ejecución. Una variable local estática sigue existiendo en el tiempo de ejecución, pero se puede eliminar de la tabla de símbolos porque no se puede acceder en tiempo de compilación una vez que está fuera del alcance).

El compilador también debe poder recorrer la lista de variables locales en el orden en que ellos fueron declarados, por ejemplo, cuando el compilador está configurando el marco de pila, tiene atravesar la lista de argumentos en el orden correcto para determinar los desplazamientos correctos de el puntero del marco, ambas situaciones se pueden manejar proporcionando un conjunto de cruz enlaces que conectan todas las variables en un nivel de anidación particular. Por ejemplo, hay tres niveles de anidación en el listado 6 [**?**]:

Listing 6: Ejemplo en C. Niveles de anidación

```
1 int Godot;
2     waiting( vladimir, estragon )
3     {
4         int pozzo ;
5         while( condition )
6         {
7         int pozzo, lucky;
8         }
9     }
```

Generación de código de 3 y 4 direcciones

Para que el análisis sintáctico sea útil cualquier cosa más allá de la mera verificación gramatical, debe producir una representación intermedia del programa que encarne lo que se descubrió en el programa. Este puede ser el código intermedio en sí mismo, o puede estar en una forma que pueda usarse como base para la generación del código intermedio. Las representaciones más comunes son árboles sintácticos, grafos acíclicos dirigidos (GAD), notación postfija y código de tres direcciones (C3D).

Código de 3 direcciones

La cuarta forma de representación intermedia es el código de tres direcciones (C3D). Este formato divide el programa en declaraciones básicas que no tienen más que tres variables y no más de un operador. La declaración $x = a + b * b$ se

traduce en las siguientes declaraciones C3D:

$$T := b * b$$

$$x := a + T$$

donde T es una variable temporal asignada para guardar el producto. Esta notación representa un compromiso; que tiene la forma general de un lenguaje de nivel superior, pero las declaraciones individuales son lo suficientemente simples como para mapear en lenguaje ensamblador de una manera razonablemente directa. Las dos declaraciones anteriores, en el lenguaje ensamblador de la familia de computadoras IBM System / 370,

```
L 3, B
M 2, B          {T:= b * b }
ST 3,T


L 3,A
A 3, T          {x:= a + T }
ST 3,X
```

o, de manera más concisa, usando el Registro General 3 para obtener resultados temporales,

```
L 3,B
MR 2,3
A 3,A
ST 3,X
```

Observe que en el nivel de la máquina, A, B, T y X son direcciones. En la generación de código intermedio, se piensa en términos de direcciones. En este punto, son aptos para ser direcciones de tabla de símbolos en lugar de direcciones de memoria, ya que generalmente no se sabe a dónde van a terminar en la memoria, pero las direcciones son, sin embargo, y esta es la razón por la cual esta notación se llama C3D. Puede ser útil pensar en C3D como el lenguaje ensamblador de tres direcciones máquina. Es decir, si se escribió nuestra segunda instrucción C3D como

$$ADD \quad X, A, T$$

se podría imaginar una computadora que ejecutó estas instrucciones para buscar el contenido de los registros A y T en la memoria y almacenar la suma en X. Cuando hablamos de C3D, usando la notación $x := y \;\; op \;\; z$, ignoremos el anuncio que se ve, pero si consideramos cómo se almacenan las variables en la memoria (o en la tabla de símbolos), tenemos que tomar en cuenta las direcciones. Las instrucciones en C3D se representan en forma de cuádruples. Un cuádruple tiene la forma

$$| \; op \; | \; addr(y) \; | \; addr(z) \; | \; addr(x) \; |$$

donde los diversos campos están en ubicaciones de memoria contigua. Ya que es problemático dibujar todos estos cuadros, es más fácil escribir

$$(op, addr(y), addr(z), addr(x))$$

El código de tres direcciones se puede generar a partir de un recorrido de un árbol de sintaxis o GAD, o puede generarse como código intermedio directamente en el transcurso del análisis. En el primer caso, el árbol de sintaxis o GAD es la representación intermedia y el código intermedio C3D; en este último caso, el C3D es a la vez la representación intermedia y el código intermedio [1].

Generación de código intermedio para estructuras iterativas y de control(for, while, if-else, switch)

Estructuras iterativas y de control

Un bloque básico es solo una secuencia lineal de longitud máxima, de código impredecible. Cualquier declaración que no afecte el flujo de control puede aparecer dentro de un bloque. Cualquier transferencia de flujo de control finaliza el bloqueo, al igual que la sentencia etiquetada ya que puede ser el objetivo de una rama. A medida que el compilador genera código, puede construir bloques básicos simplemente agregando consecutivos, no etiquetados, operaciones que no sean flujo de control. (Se supone que una declaración etiquetada no esta etiquetada gratuitamente, es decir, cada declaración etiquetada es el objetivo de alguna rama.) La representación de un bloque básico no necesita ser compleja. Por ejemplo, si el compilador tiene una representación similar al ensamblador que se tiene en un arreglo lineal, luego un bloque puede describirse por un par,$\langle first, last \rangle$, que contiene los índices de la instrucción donde comienza el bloque y la instrucción que termina el bloque. (Si los índices de bloques se almacenan en orden numérico ascendente, una serie de primeros bastará). Para unir un conjunto de bloques para que formen un procedimiento, el compilador debe insertar un código que implemente las operaciones de control de flujo del programa fuente. Para capturar las relaciones entre bloques, muchos compiladores construyen un gráfico de control de flujo (GCF) y lo usa para analizar y así realizar, la optimización y generación de código. En el GCF, los nodos representan bloques básicos y los bordes representan posibles transferencias de control entre bloques. Típicamente, el GCF es una representación derivada que contiene referencias a una representación más detallada de cada bloque. El código para implementar la construcción del control de flujo que reside en los bloques básicos o cerca del final de cada bloque. (En ensamblador, no hay caso de caída en una rama, por lo que cada bloque termina con una rama o un salto. Si el modelo de representación intermedia retraza la ranura, entonces la operación de control de flujo puede no ser la última operación en el bloque). Si bien se han

utilizado muchas convenciones sintácticas diferentes para expresar el control de flujo, la cantidad de conceptos subyacentes es pequeña [5].

Ejecución condicional

La mayoría de los lenguajes de programación proporcionan alguna versión para construir un **if-then-else**. Dado el texto fuente del listado 7 .

Listing 7: Ejemplo if-else

```
1        if expresion
2            sentencia 1
3        else sentencia 2
4            sentencia 3
```

el compilador debe generar código que evalúe la **expresión** y las ramas para la **sentencia 1** o **sentencia 2**, basado en el valor de **expresión**. El código ensamblador que implementa las dos declaraciones debe terminar con un salto a la **sentencia 3**. El compilador tiene muchas opciones para implementar construcciones **if-then-else**.

Los programadores pueden colocar fragmentos de código arbitrariamente grandes dentro de otras partes. El tamaño de estos fragmentos de código tiene un impacto en la la estrategia de empilar para implementar la construcciones **if-then-else**. Con una parte trivial **then** y **else**, la consideración principal para el compilador es hacer coincidir la evaluación de la expresión con el hardware subyacente. Como **then** y **else** crecen, la importancia de una ejecución eficiente dentro de **then** y **else** entonces, comienza a superar el costo de ejecutar el control de la expresión.

Como se muestra en el listado 8, en una máquina que admite la ejecución predicada, usando predicados en los bloques grandes de **then** y **else** pueden desperdiciar ciclos de ejecución. Dado que el procesador debe emitir cada instrucción predicada a una de sus unidades funcionales, cada operación con un predicado falso tiene un costo de oportunidad alta

en un puesto de edición. Con grandes bloques de código tanto en **then** y **else**, el costo de las instrucciones no ejecutadas puede superar el costo general de usar una rama condicional.

Se supone que tanto **then** y **else** contienen 10 operaciones iloc independientes y que la máquina objeto puede emitir dos operaciones por ciclo.

El código que podría generarse mediante predicado; eso asume que el valor de la expresión de control está en r_1. El código emite dos instrucciones por ciclo. Uno de ellos se ejecuta en cada ciclo. Luego las operaciones de la parte **then** se envían a la **Unidad 1**, mientras que la operación de la parte **then** se envían a la **Unidad 2**. El código evita todas las ramificaciones. Si cada operación toma un solo ciclo, lleva 10 ciclos ejecutar las declaraciones controladas, independiente de qué rama se tome.

El listado 9 muestra el código que podría generarse usando ramas; se asume ese control y fluye a L_1 para la parte **then** o a L_2 para la parte **else**. Porque las instrucciones son independientes, el código emite dos instrucciones por ciclo. Siguiendo la ruta entonces toma cinco ciclos para ejecutar las operaciones para la ruta tomada, más el costo del salto terminal. El costo de la parte else es idéntico. La versión predicada evita la rama inicial requerida en la versión no

programada del código (a L_1 o L_2 en la figura), así como a los saltos terminales (a L_3). La versión de ramificación incurre en la sobrecarga de una rama y un salto, pero puede ejecutar más rápido. Cada ruta contiene una rama condicional, cinco ciclos de operaciones, y el salto terminal. (Algunas de las operaciones se pueden usar para llenar ranuras de retraso en los saltos). La diferencia radica en la tasa de emisión efectiva, la versión de ramificación emite aproximadamente la mitad de las instrucciones de la versión de predicado. A medida que el código se fragmenta en el **else**, esto hace que la diferencia se haga más grande. Elegir entre ramificación y predicación para implementar un **if-then-else** requiere un poco de cuidado. Varios problemas deben ser considerados, como sigue:

1. Frecuencia esperada de ejecución. Si se ejecuta un lado del condicional significativamente más a menudo, las técnicas que aceleran la ejecución de esa ruta puede producir código más rápido. Este sesgo puede tomar la forma de predecir una rama, al ejecutar algunas instrucciones especulativamente, o de reordenar la lógica.

2. Cantidades desiguales de código si una ruta a través de la construcción contiene muchas más instrucciones que la otra, esto puede pesar contra predicación o para una combinación de predicación y ramificación.

3. Control del flujo dentro de la construcción. Si cualquiera de las rutas contiene algún flujo de control no trivial, como un bucle **if-then-else**, declaración de caso, o llamar, entonces la predicación puede ser una mala elección. En particular, anidado si constructos crean predicados complejos y reducen la fracción de operaciones emitidos que son útiles.

Para tomar la mejor decisión, el compilador debe considerar todos estos factores, así como el contexto circundante. Estos factores pueden ser difíciles de evaluar al principio de la compilación; por ejemplo, la optimización puede cambiarlos de manera significativa [5].

Listing 8: Uso de predicados

```
1  Unidad 1              Unidad 2
2
3      comparacion ⇒ r1
4
5  (r1) op1                  ( ¬ r1 ) op11
6  (r1) op2                  ( ¬ r1 ) op12
7  (r1) op3                  ( ¬ r1 ) op13
8  (r1) op4                  ( ¬ r1 ) op14
9  (r1) op5                  ( ¬ r1 ) op15
10 (r1) op6                  ( ¬ r1 ) op16
11 (r1) op7                  ( ¬ r1 ) op17
12 (r1) op8                  ( ¬ r1 ) op18
13 (r1) op9                  ( ¬ r1 ) op19
14 (r1) op10                 ( ¬ r1 ) op20
```

Listing 9: Uso de ramas

```
1      Unidad 1              Unidad 2
```

```
2
3       comparacion & rama
4
5   L1 :        op1     op2
6               op3     op4
7               op5     op6
8               op7     op8
9               op9     op10
10              saltoI → L3
11
12
13  L2 :        op11    op12
14              op13    op14
15              op15    op16
16              op17    op18
17              op19    op20
18              saltoI → L3
19
20
21  L3 :        nop
```

Condicional for

Para mapear un bucle **for** en el código, el compilador sigue el esquema general del listado 10 . Para hacer esto considere el siguiente ejemplo. Pasos 1 y 2 producen un solo bloque básico, como se muestra en la segunda parte del listado 10 :

Listing 10: Código ensamblador para el **for** en C

```
1  Primera parte
2
3  for (i=1; i<=100; i++) {
4  cuerpo del ciclo
5  }
6  siguiente sentencia
7
8
9  Segunda parte
10
11
12      loadI 1 ⇒ ri // Paso 1
13      loadI 100 ⇒ r1 // Paso 2
14      cmp GT ri , r1 ⇒ r2
15      cbr r2 → L2 , L1
16 L1 : cuerpo del ciclo // Paso 3
17
18      addI ri , 1 ⇒ ri // Paso 4
19      cmp LE ri , r1 ⇒ r3
20      cbr r3 → L1 , L2
```

```
21 L2 : siguiente sentencia // Paso 5
```

El código producido en los pasos 1, 2 y 4 es sencillo. Si el cuerpo del bucle (paso 3) consiste en un solo bloque básico o termina con un solo básico bloquear, entonces el compilador puede optimizar la actualización y la prueba producidas en el paso 4 con el cuerpo del bucle Esto puede conducir a mejoras en el código, por ejemplo, ple, el programador de instrucciones podría usar operaciones desde el final del paso 3 hasta que llene las ranuras de retardo en la rama del paso 4. El compilador también puede dar forma al ciclo para que tenga solo una copia de la prueba del paso 2. De esta forma, el paso 4 evalúa e 3 y luego salta al paso 2. El compilador reemplazaría cmp LE, secuencia cbr al final del ciclo con un salto I. Esta forma del ciclo es una operación más pequeña que las dos: forma de prueba. Sin embargo, crea un bucle de dos bloques incluso para los bucles más simples, y alarga la ruta a través del ciclo por al menos una operación. Cuando el tamaño del código es una consideración seria, el uso constante de este bucle más compacto la forma puede valer la pena. Siempre que el salto que termina en bucle sea inmediato saltar, el hardware puede tomar medidas para minimizar cualquier interrupción que pueda porque. La forma del bucle canónico del listado 11 también prepara el escenario para la optimización. Por ejemplo, si e_1 y e_2 contienen solo constantes conocidas, como en el listado 10 , el compilador puede duplicar el valor del paso 1 en la prueba del paso 2 elimina la comparación y la bifurcación (si el control ingresa al loop) o elimina el cuerpo del loop (si el control nunca entra en el loop). En el bucle de prueba única, el compilador no puede hacer esto. En cambio, el compilador encuentra dos caminos que conducen a la prueba: uno desde el paso 1 y uno desde el paso 4. El valor utilizado en la prueba, r_i, tiene un valor variable a lo largo del borde desde el paso 4, por lo que el resultado de la prueba no es predecible [5] .

Condicional while

Un ciclo **while** también se puede implementar con el esquema de bucle del listado 11 . A diferencia del bucle en C para **for** o un **do** de fortran, un bucle **while** no tiene inicialización, por lo tanto, el código es aún más compacto como se muestra en el listado 11 .

Listing 11: Código ensamblador para el **while**

```
1  while (x < y) {
2  cuerpo del ciclo
3  }
4  siguiente sentencia
5
6
7
8
9      cmp LT rx , ry ⇒ r1 // Paso 2
10     cbr r1 → L1 , L2
11
12 L1 : cuerpo del ciclo // Paso 3
13     cmp LT rx , ry ⇒ r2 // Paso 4
14     cbr r2 → L1 , L2
```

```
15 L2 : siguiente sentencia // Paso 5
```

Replicar la prueba en el paso 4 crea la posibilidad de un ciclo con un solo bloque básico. Los mismos beneficios que se acumulan en un bucle **for** desde esta estructura también ocurren durante un ciclo **while** [5] .

Listing 12: Código ensamblador para el **do** de Fortran

```
1  j = 1
2  do 10 i = 1, 100
3
4      cuerpo del ciclo
5
6  j = j + 2
7
8  10 continue
9
10     siguiente sentencia
11
12
13
14
15     loadI 1 ⇒ rj // j ← 1
16     loadI 1 ⇒ ri // Paso 1
17     loadI 100 ⇒ r1 // Paso 2
18     cmp GT ri , r1 ⇒ r2
19     cbr r2 → L2 , L1
20
21 L1 : cuerpo del ciclo // Paso 3
22     addI rj , 1 ⇒ rj // j ← j + 2
23     addI ri , 1 ⇒ ri // Paso 4
24     cmp LE ri , r1 ⇒ r3
25     cbr r3 → L1 , L2
26 L2 : siguiente sentencia // Paso 5
```

Condicional switch

Muchos lenguajes de programación incluyen alguna variante de una declaración de **case**. Fortran tiene su **goto** calculado. Algol-W presentó la declaración de caso en su forma moderna. bcpl y C tienen una construcción de interruptor, mientras que pl / i tiene una construcción generalizada que se correlaciona bien con un conjunto anidado de declaraciones **if-then-else**. Como lo insinuó la introducción a este capítulo, implementar un caso de declaración de manera eficiente es compleja. Considere la implementación de la sentencia **switch** de C. La estrategia básica es sencilla:

1. evaluar la expresión de control;
2. rama al **case** seleccionado;
3. ejecutar el código para ese **case**.

Los pasos 1 y 3 están bien entendidos, como se desprende de las discusiones en otro lugar de este capítulo. En C, los **case's** individuales generalmente terminan con una declaración de **break** que sale de la declaración **switch**. La parte compleja de la implementación del enunciado case reside en elegir un método eficiente para localizar el case designado. Porque el **case** deseado es que no se conoce hasta el tiempo de ejecución, el compilador debe emitir código que utilizará el valor de la expresión de control para ubicar el correspondiente **case**. No solo el método funciona bien para todas las declaraciones del **case**. Muchos compiladores tienen una disposición para varios esquemas de búsqueda diferentes y eligen entre ellos en función de los detalles específicos del conjunto de **case's**. Esta sección examina tres estrategias: una búsqueda lineal, una búsqueda binaria y una dirección calculada de cada estrategia que es apropiada bajo diferentes circunstancias y posturas [5].

Generación de código intermedio para llamado a funciones y procedimientos

La implementación de llamadas a procedimientos es, en su mayor parte, directa. Como se muestra en la figura 7.19, una llamada de procedimiento consiste en una secuencia previa a la llamada y una secuencia post return en la persona que llama, y un prólogo y un epílogo en el llamado. Un único procedimiento puede contener múltiples sitios de llamadas, cada uno con sus propias secuencias previas y posteriores al cambio. En la mayoría de los idiomas, un procedimiento tiene un punto de entrada, por lo que tiene una secuencia de prólogo y una secuencia de epílogo. (Algunos idiomas permiten puntos de entrada múltiples, cada uno de los cuales tiene su propia secuencia de prólogo.) Muchos de los detalles involucrados en estas secuencias se describen en la Sección 6.5. Esta sección se enfoca en asuntos que afectan la capacidad del compilador para generar código eficiente, compacto y consistente para llamadas de procedimiento [5].

Manejo del polimorfismo en un compilador

Algunas funciones se ejecutan de una manera que es independiente del tipo de datos en el que ellas operan. Algunas estructuras de datos están formadas de la misma manera independientemente de los tipos de sus elementos. Como ejemplo, considere una función para concatenar dos listas enlazadas en Tiger. Primero se definen un tipo de datos de la lista enlazada y luego la función de concatenación:

DEFINICIÓN 4 (Polimorfismo.) *Una función que puede operar con argumentos de diferentes tipos es una función polimórfica. Si el conjunto de tipos debe especificarse explícitamente, la función usa polimorfismo ad hoc; Si el cuerpo de la función no especifica tipos, utiliza polimorfismo paramétrico [5].*

No hay nada sobre el código para el tipo de datos **int list** o el apéndice de la función que sería diferente si el tipo de elem fuera de la cadena o en lugar del árbol. Nos gustaría añadir para poder trabajar en cualquier tipo de lista. Una función polimórfica (del griego many + shape) si puede operar en argumentos de diferentes tipos. Hay dos tipos principales de polimorfismo:

1. Polimorfismo paramétrico. Una función es paramétricamente polimórfica si sigue baja el mismo algoritmo, independientemente del tipo de su argumento. El Ada o El mecanismo genérico Modula-3, las plantillas de C ++ o los esquemas de tipo ML son examinados ples de polimorfismo paramétrico.

2. Sobrecarga o polimorfismo ad hoc. Un identificador de función está sobrecargado si representa diferentes algoritmos dependiendo del tipo de su argumento. Por ejemplo, en la mayoría de los idiomas el signo + está sobrecargado, lo que significa que la suma entera en argumentos enteros y flotantes la adición de punto (que es un algoritmo bastante diferente) en argumento de coma flotante. En muchos idiomas, incluidos Ada, C ++ y Java, los programadores pueden hacer funciones sobrecargadas propias. Estos dos tipos de polimorfismo son bastante diferentes, casi no relacionados, y requieren diferentes técnicas de implementación [**?**].

Algunos idiomas permiten que una función sea polimórfico o genérico, es decir, definido en una gran clase de tipos, por ejemplo, en todas las matrices sin importar cuáles sean los tipos de elementos. Una función puede declarar explícitamente qué partes del tipo es genérico / polimórfico o esto puede ser implícito. El verificador de tipos puede insertar el tipo real en cada uso de la función genérica / polimórfica para crear instancias del tipo genérico / polimórfico. Este mecanismo es diferente de la sobrecarga, ya que las instancias estarán relacionadas por un tipo genérico común y porque una función polimórfica / genérica se puede instanciar por cualquier tipo, no solo por un límite de lista de alternativas declaradas como es el caso de la sobrecarga [4].

Ejercicios

E1.- Genérese código para las siguientes proposiciones en C.

1. x = f(a) + f(a) + f(a)

2. x = f(a)/a(b,c)

3. x = f(f(a))
4. x = ++f(a)
5. *p++ = *g++

E2.- Genérese código para el siguiente programa en C

```
1  main()
2      {
3          int i;
4          int a[10];
5          while (i <= 10)
6              a[i] = 0;
7      }
```

E3.- Genérese código para las siguientes proposiciones en C

1. x = 1
2. x = y
3. x = x + 1
4. x = a + b * c
5. x = a/(b + c) - d*(e + f)

E4.- Genérese código para el siguiente programa en Pascal:

```
1  programa lazofor(input, output);
2      var i, inicial, final: integer;
3      begin
4          read(inicial, final);
5          for i := inicial to final do
6              writels(i)
7      end
```

E5.- Construir el traductor dirigido por la sintaxis de la siguiente gramática:

1. $N \rightarrow SL$
2. $S \rightarrow +|-$
3. $L \rightarrow LB|B$
4. $B \rightarrow 0|1$

Optimización de código

A las flores les pedimos que tengan perfume. A los hombres, educación

Proverbio inglés

El código generado por la traducción dirigida por sintaxis tiende a contener una gran cantidad de código muerto. Cuando un analizador produce C3D como

$$T := a + b$$

$$x := T$$

que claramente se puede simplificar a

$$x := a + b$$

Pero hay instrucciones redundantes de todo tipo, y con frecuencia las operaciones son implementadas de manera torpe que ningún programador en lenguaje ensamblador experimentado usaría. De hecho, el mejor código sería el producido por una asamblea experta: programador de idiomas trabajando con una máquina familiar en un problema bien entendido. Este es un ideal que ningún compilador puede lograr. La optimización es el intento de alcanzar este ideal, el problema del optimizador es eliminar la mayor parte de este código muerto como sea posible sin cambiar el significado del programa. Probablemente sería mejor llamarlo "mejorar el código", ya que la palabra "optimización" sugiere que el código resultante será óptimo, es decir, ideal. Sin embargo, la optimización es el término aceptado y es el que se usará.

Hay límites en cuanto a lo que la optimización puede hacer, porque lo que hace sin sentido a un programa, es que no puede ejercer ningún juicio y que no debe hacer nada, lo que cambia el significado del programa. De hecho, la optimización puede en raras ocasiones hacer que un programa sea más grande o más lento de lo que era antes. Además, algunos de los compiladores omiten la optimización por completo. Un compilador para usarse en la enseñanza de la programación no necesita producir código eficiente. Esto se debe a que la mayor parte del tiempo de un estudiante se gastará en la depuración; el programa normalmente se volverá a compilar cada vez que es ejecutado. Una vez que el programa sea correcto, el alumno lo entregará y nunca más volverá a ejecutarlo. En tal compilador, el manejo de errores es útil y muy deseable, por lo que la optimización se hace innecesaria.

Una función del sistema operativo o una utilidad del subprograma como un género, por otro lado, se ejecutarán decenas o cientos de miles de veces durante su vida útil, probablemente por muchas personas y probablemente como repite parte de los programas largos y lentos, y en este caso la optimización es esencial.

Por otra parte, la optimización no es un sustituto para un buen diseño del programa o especialmente para seleccionar el algoritmo inteligente. Ninguna cantidad de optimización tendrá un efecto significativo o aceleración de un orden O(n^2); la solución aquí es ir a un algoritmo O(n lg n) como ordenación rápida. A pesar de estas advertencias, la optimización fue un problema desde el principio, porque los primeros compiladores tuvieron que demostrar su valía con los programadores que ya eran adeptos a la programación en lenguaje ensamblador, y con el empleo de este, los programadores hicieron una gran cantidad de mejoras que se pueden hacer en la salida de la generación de código intermedio. Por esta razón, y porque la optimización se ha estudiado durante tanto tiempo, ahora hay un enorme cuerpo de técnicas de optimización.

Las técnicas conocidas, y los métodos para implementarlas.

La mayor parte de la optimización consiste en identificar cálculos innecesarios. Un cálculo es innecesario si calcula algo que ya se conoce; por lo tanto, la necesidad de encontrar una forma de identificar los valores de las variables o subexpresiones que son conocidas. Se ha visto que los GDA se pueden usar para este propósito, en pocas palabras para encontrar las secuencias de instrucciones. Para un análisis a mayor escala, debemos tener una forma de rastreo del modo en que la información sobre los elementos de datos se mueven a través del programa; esto es conocido como análisis de flujo de datos, y es una herramienta principal en la optimización [1].

Introducción de código

La finalidad de la optimización de código es producir un código objeto lo más eficiente posible. En algunos casos también se realiza una optimización del código intermedio. Algunas optimizaciones que se pueden realizar son la evaluación de expresiones constantes, el uso de ciertas propiedades de los operadores, tales como la asociativa, conmutativa, y distributiva; así como la reducción de expresiones comunes [**?**].

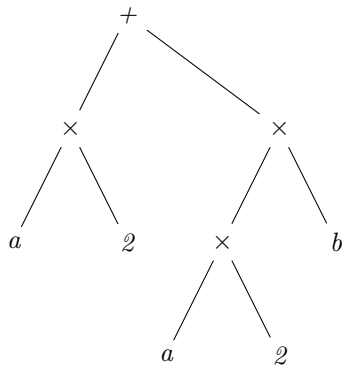
Considerando que el código que entra no es optimo, el GDA se aplica para encontrar las secuencias de instrucciones y el rastreo del modo en que la información sobre los elementos de datos se mueven a través del programa por medio del análisis del flujo de datos. Todo esto con la finalidad de mejorar el código de entrada. Así que para definir el GDA se define primero el Árbol de Sintaxis Abstracta (ASA):

DEFINICIÓN 5 (Árbol de Sintaxis Abstracta.) *Un ASA es una contracción del árbol de análisis sintáctico que omite la mayoría de los nodos para símbolos no terminales [5].*

DEFINICIÓN 6 (Grafo Dirigido Acíclico.) *Un GDA es un ASA con intercambio. Subárboles idénticos son instanciados una vez, con múltiples padres [5].*

El árbol de sintaxis abstracta (ASA) conserva la estructura esencial del árbol de análisis sintáctico pero elimina los nodos extraños. La precedencia y el significado de la la expresión permanece, pero los nodos extraños han desaparecido. Aquí está el ASA para a × 2 + a × 2 × b:

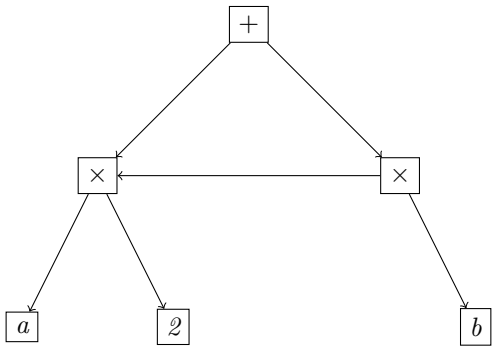
EJEMPLO 1 *El ASA de la expresión* $a \times 2 + a \times 2 \times b$*, es:*



Mientras que el ASA es más conciso que un árbol de sintaxis, conserva fielmente el estructura del código fuente original. Por ejemplo, el ASA para $a \times 2 + a \times 2 \times b$ contiene dos copias distintas de la expresión $a \times 2$. Un gráfico dirigido acíclico (GDA) es una contracción del ASA que evita esta duplicación. En un GDA, los nodos pueden tener múltiples padres e idénticos subárboles que son reutilizados. Tal intercambio hace que el GDA sea más compacto que el ASA correspondiente.

Para las expresiones sin asignación, expresiones textualmente idénticas deben producir valores idénticos. El GDA para $a \times 2 + a \times 2 \times b$, se muestra en el EJEMPLO 2, que refleja el hecho de compartir una sola copia de un × 2. El GDA codifica una sugerencia explícita para evaluar la expresión. Si el valor no se puede cambiar entre los dos usos de a, entonces el compilador debe generar código para evaluar $a \times 2$ una vez y debe usar el resultado dos veces. Esta estrategia puede reducir el costo de la evaluación, sin embargo, el compilador debe probar que el valor puede ser: que la expresión no contiene ninguna asignación ni llamadas a otra procedimientos, la prueba es fácil. Dado que una asignación o una llamada de procedimiento puede cambiar el valor asociado con un nombre, el algoritmo de construcción GDA debe invalidar los subárboles a medida que cambian los valores de sus operandos.

EJEMPLO 2 *El GDA de la expresión* $a \times 2 + a \times 2 \times b$*, es:*



Análisis de flujo de datos

Tan pronto como se expanda el alcance de la optimización más allá del bloque básico, debemos ser capaz de rastrear cómo los valores se abren paso a través de un programa. Este proceso se llama análisis de flujo de datos [1].

DEFINICIÓN 7 (Grafo de dependencia de datos.) *Es un gráfico que modela el flujo de valores de definiciones con usos en un fragmento de código [5] .*

Los compiladores también usan gráficos para codificar el flujo de valores desde el punto donde se crea un valor, una definición, en cualquier punto donde se usa. Un grafo de dependencia encarna esta relación. Los nodos en un grafo de dependencia de datos representan operaciones. La mayoría de las operaciones contienen definiciones y usos. Un borde en un grafo de dependencia de datos conecta dos nodos, uno que define un valor y otro que lo usa. Dibujamos grafos de dependencia con bordes que se ejecutan desde la definición hasta el uso [5] .

Considere el fragmento de código que se muestra en el listado 13 . Las respuestas a **a[i]** se muestran derivando sus valores de un nodo que representa definiciones de **a**. Esto conecta todos los usos de un conjunto a través de un solo nodo. Sin un sofisticado análisis de las expresiones de subíndices, el compilador no puede diferenciar entre referencias a elementos de las matrices individuales. Los nodos 5 y 6 ambos dependen de sí mismos; usan valores que pueden ser definidos en una iteración anterior. El nodo 6, por ejemplo, puede tomar el valor de 2 (en la iteración inicial) o de sí mismo (en cualquier iteración). Los nodos 4 y 5 también tienen dos fuentes distintas para el valor de **i**: nodos 2 y 6.

Los grafos de dependencias de datos a menudo se usan como derivados de la construcción a partir de una tarea específica, usada y luego descartada. Juegan

un papel central en la programación de instrucciones. Encuentran aplicación en una variedad de optimizaciones, particularmente transformaciones que reordenan bucles para exponer el paralelismo y mejorar el comportamiento de la memoria; estos típicamente requieren un análisis sofisticado de subíndices de la matriz para determinar con mayor precisión los patrones de acceso a las matrices. En aplicaciones más sofisticadas de los datos del grafo de dependencia, el compilador puede realizar un análisis exhaustivo de la matriz de valores de subíndice para determinar cuándo las referencias a la misma matriz pueden superponerse [5] .

Listing 13: Flujo de control

```
1 x ← 0
2 i ← 1
3 while (i < 100)
4     if (a[i] > 0)
5         then x ← x + a[i]
6     i ← i + 1
7 print x
```

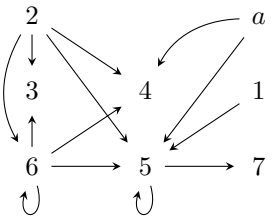


Figura 3: El grafo de dependencias del listado 13 .

DEFINICIÓN 8 (Bloque básico.) *Un bloque básico* $B = (s_1, ..., s_n)$ *es una secuencia de declaraciones de representación intermedia de longitud máxima, para las cuales se cumplen las siguientes condiciones: B solo se ingresa en la declaración* s_1 *y se deja en* s_n*. El enunciado* s_1 *se llama el líder del bloque básico. Puede ser un punto de entrada de la función, un destino del salto o una declaración que sigue inmediatamente después de un salto o un retorno [?].*

Listing 14: Bloque básico en un código de tres direcciones.

```
1 t1 := a * a
2 t2 := a * b
3 t3 := 2 * t2
4 t4 := t1 + t3
5 t5 := b * b
6 t6 := t4 + t5
```

ALGORITMO 1 (Partición en bloques básicos.)
Entrada. *Un secuencia de proposiciones de tres direcciones.*

Salida. *Una lista de bloques básicos donde cada proposición de tres direcciones está en un bloque exactamente.*

Método.

1. *Primero se determina el conjunto de lideres, la primera proposición de cada bloque básico. Las reglas que se utilizan son las siguientes:*
   a) *La primera proposición es un líder.*
   b) *Cualquier proposición que sea el destino de un salto* ***goto*** *condicional o incondicional es un líder.*
   c) *Cualquier proposición que vaya inmediatamente después de un salto* ***goto*** *condicional o incondicional es un líder.*
2. *Para cada lider, su bloque básico consta del líder y de todas las proposiciones hasta, pero sin incluir, el siguiente líder o el fin del programa.*

[3].

Las optimizaciones locales se realizan sobre el bloque básico. Un bloque básico es un fragmento de código que tiene una única entrada y salida, y cuyas instrucciones se ejecutan secuencialmente. Implicaciones: si se ejecuta una instrucción del bloque se ejecutan todas en un orden conocido en tiempo de compilación. La idea del bloque básico es encontrar partes del programa cuyo análisis es necesario para que la optimización sea lo más simple posible [**?**].

DEFINICIÓN 9 (Grafo de flujo.) *Una representación de grafos de proposiciones de tres direcciones, llamada grafo de flujo, es útil para entender los algoritmos de generación de código, incluso aunque el grafo no esté explicitamente construido por un algoritmo de construcción de código. Los nodos del grafo de flujo representan cálculos y las aristas representan el flujo de control [3] .*

EJEMPLO 3 (Grafo de flujo.) *Considerese el programa que aparece en el listado 15 y en listado 16 el mismo programa en código de tres direcciones: en la figura 4 se representa su grafo de flujo [3] .*

Listing 15: Programa para calcular el producto punto.

```
1  begin
2      prod i := 0;
3      i := 1;
4      do begin
5          prod := prod + a[i] * b[i];
6          i := i + 1;
7      end
8      while i <= 20
9  end
```

Listing 16: Código de tres direcciones para calcular el producto punto.

```
1  prod := 0
2  i := 1
3  t1 := 4 * i
4  t2 := a[t1]
5  t3 := 4 * i
6  t4 := b[t3]
7  t5 := t2 * t4
8  t6 := prod + t3
9  prod := t6
10 t7 := i + 1
11 i := t7
12 if i <= 20 goto (3)
```

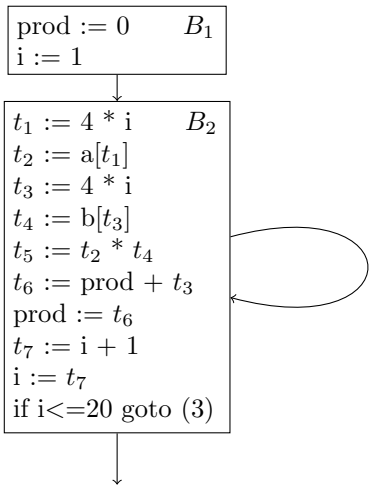


Figura 4: El grafo de flujo del listado 16 .

DEFINICIÓN 10 (Grafo de flujo de datos, GFD.) *Un grafo de flujo de datos (GFD) para un bloque básico B es un grafo dirigido acíclico $G_B = (V_B, E_B)$, donde cada nodo $v \in V_B$ representa un operando de entrada (constante, variable), un operando de salida (variable) o una operación en representación intermedia. Un arco $e = (v_i, v_j) \in E_B \subset V_B \times V_B$ indica que el valor definido por v_i es utilizado por v_j [?].*

Un GFD se denomina árbol de flujo de datos (AFD) si ningún nodo tiene más de una transición saliente, es decir, no hay subexpresiones comunes. Por lo general, las AFD construyen los datos de entrada de acuerdo a muchas técnicas populares de selección de código. En la práctica, los compiladores realizan un AFD para una función completa, llamada AFD global, dado que un AFD local dificulta muchas oportunidades de optimización. Supongamos, un bloque básico que tiene varias transiciones salientes de flujo de control, es decir, una definición de una variable (por ejemplo, una constante) puede alcanzar múltiples usos, posiblemente en diferentes bloques básicos. Por lo tanto, para explotar todo el potencial de la propagación constante, se requieren todos los usos alcanzados por esas definiciones, que solo pueden ser proporcionados por un AFD. Por lo general, AFD local está integrado como una subrutina en el AFD global que iterativamente resuelve las ecuaciones de flujo de datos para un procedimiento completo. El análisis puede extenderse incluso más allá de los límites de la función. La idea general detrás de un llamado análisis interprocedimental es recoger la información que fluye en una función y luego usarla, para actualizar la información local. Esto requiere información acerca de [**?**] :

1. que función f_t a cualquier función particular f llama,
2. Los valores de retorno de la función f,
3. qué funciones f_c llaman a cualquier función particular g, y
4. qué argumentos de f_c pasan a la función g.

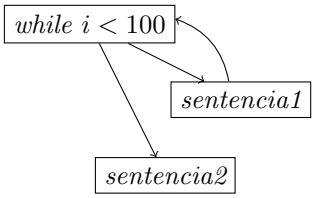
DEFINICIÓN 11 (Grafo de flujo de control, GFC.) *Un grafo de flujo de control (GFC) de una función F es un grafo dirigido $G_F = (V_F, E_F)$. Cada nodo $v \in V_F$ representa un bloque básico, y E_F contiene un arco (v, v') $\in V_F \times V_F$ si v' se puede ejecutar directamente después de v. El conjunto de sucesores de un bloque básico B está dado por $\boldsymbol{succ}_B = \{v \in V_F | (b, v) \in E_F\}$ y el conjunto de predecesores de un bloque básico B está dado por $\boldsymbol{pred}_B = \{v \in V_F | (v, b) \in E_F\}$ [?].*

El GFC proporciona una representación gráfica de la posible trayectoria del control de flujo en tiempo de ejecución. El GFC difiere de las representaciones intermedias orientadas a la sintaxis, como un ASA, en que los arcos muestran la estructura gramatical. Considere el código del listado 17 , en el **EJEMPLO 4** se representa el GFC para el ciclo while [5] :

Listing 17: Código de un ciclo while.

```
1 while(i < 100)
2     begin
3      sentencia1
4      end
5 sentencia2
```

EJEMPLO 4 *Grafo de flujo de control del ciclo while:*



El arco de la **sentencia1** regresa hacia el encabezado del bucle para crear un ciclo; el ASA para este fragmento sería acíclico.

Propagación de constantes

Para un programador relativamente novato, puede parecer algo paradójico tener la capacidad de declarar una constante; una constante es constante (es decir, no cambia), entonces, ¿por qué declararla? Si bien es cierto que las constantes no pueden ser modificadas durante una ejecución particular de un programa, es un activo importante el ser capaz de definir una constante como **symbsize** (significado del tamaño de la tabla de símbolos) una vez y usarlo varias veces en el programa. Haciendo un cambio a una constante solamente implica cambiar la declaración. Este cambio se reflejará en cualquier lugar donde la constante declarada se usa; por ejemplo, un cambio en **symbsize** puede tener el efecto de cambiar el tamaño de la tabla de símbolos y alterar todos los controles que están hechos para el desbordamiento de la tabla de símbolos en un compilador. En la mayoría de los lenguajes de programación con declaraciones constantes, estas pueden ser solo declaradas una vez en cualquier módulo separadamente compilable. Las constantes son vistas como son definidas en el nivel global y, por lo tanto, se almacenan en la (a veces llamada la parte “constante”) tabla de símbolos [7] .

DEFINICIÓN 12 (Ensamblamiento.) *Es la sustitución de expresiones que pueden evaluarse en tiempo de compilación por sus valores calculados. Es decir, si los valores de todos los operandos de un expresión son conocidos por el compilador en tiempo de compilación, la expresión puede ser reemplazada por su valor calculado [7] .*

La propagación de constante es simplemente el ensamblamiento aplicado de tal manera que los valores conocidos en tiempo de compilación se asocian con las variables que se utilizan para reemplazar ciertos usos de estas variables en el texto del programa traducido. Por lo tanto, el fragmento del programa escrito (tal vez para mejorar la claridad) como:

Desde que se asigna a una variable un valor constante hasta la siguiente asignación, se considera a la variable constante.

EJEMPLO 5 *Propagación del Ensamblamiento:*
PI=3.14 → PI=3.14 → PI=3.14
G2R=PI/180 → G2R=3.14/180 → G2R=0.017

PI y G2R se consideran constantes hasta la próxima asignación. Estas optimizaciones permiten que el programador utilice variables como constantes sin introducir ineficiencias. Por ejemplo en C no hay constantes y será lo mismo utilizarlas como se muestra en el listado 18 :

Listing 18: Declaración de constantes

```
1 int a=10;
2 #define a 10
```

Con la ventaja de que la variable puede ser local. Actualmente en C la constante se puede definir como **const int a=10** [**?**] .

Eliminación de redundancia parcial

Un cálculo $x + y$ es redundante en algún punto **p** del código si, a lo largo de cada camino que alcanza **p**, $x + y$ ya ha sido evaluado y x y y no han sido modificados desde la evaluación. Normalmente surgen cálculos redundantes como artefactos de traducción u optimización. Hay tres técnicas efectivas para eliminar la redundancia: numeración del valor local (nvl), numeración del valor superlocal (nvs) y movimiento de código perezoso (mcp). Estos algoritmos cubren el rango de simple y rápido (nvl) a complejo y comprensivo (mcp). Si bien los tres métodos difieren en el alcance que cubren, la principal distinción entre ellos radica en el método que utilizan para establecer que dos valores son idénticos. Existe otra técnica para eliminar la redundancia, llamada técnica basada en dominantes [5] .

Numeración del valor local

Considere el bloque básico de cuatro instrucciones que se muestra en el listado 19 . Nos referiremos al bloque B. Donde la expresión, **b + c** o **a - d**, es redundante en B si y solo si se ha calculado previamente en B y ninguna operación intermedia redefine uno de sus argumentos constituyentes. En B, la ocurrencia de **b + c** en la quinta operación no es redundante porque la tercera operación redefine **b**. La aparición de **a - d** en la cuarta operación es redundante porque B no redefine **a** o **d** entre la cuarta y la sexta operación.

Listing 19: Bloque básico de cuatro instrucciones

```
1  Bloque original (B)
2
3  a ← b + c
4  b ← a - d
5  c ← b + c
6  d ← a + d
7
8
9  Bloque reescrito
10
11 a ← b + c
12 b ← a - d
13 c ← b + c
14 d ← b
```

El compilador puede reescribir el bloque B para que calcule **a - d** una vez, como se muestra en el listado 19 . La segunda evaluación de **a - d** se reemplaza con una copia de **b**. Una estrategia alternativa reemplazaría los usos posteriores de **d** con usos de **b**. Sin embargo, ese enfoque requiere un análisis para determinar si **b** es redefinido antes de algún uso de **d**. En la práctica, es más simple tener al optimizador para que inserte una copia y deje que un pase posterior determine qué operaciones de copia se hacen, de hecho, necesarias y cuáles pueden tener su origen y destino nombres combinados.

DEFINICIÓN 13 (Bloque básico extendido, BBE.) *Es un conjunto de bloques $\beta_1, \beta_2, \cdots, \beta_n$ donde β_1 tiene múltiples GFC predecesores y entre sí β_i tiene solo uno, que es algo β_j en el conjunto [5] .*

Numeración del valor superlocal

Para mejorar los resultados de la numeración de valores locales, el compilador puede extender su alcance de un solo bloque básico a un bloque básico extendido, o reflujo. A procesar un reflujo, el algoritmo debe valorar el número de cada camino a través del reflujo. Considere, por ejemplo, el código que se muestra en el listado 20 . Su GFC, se muestra en la figura 5 , contiene un reflujo no trivial, $(B_0, B_1, B_2, B_3, B_4)$ y dos triviales BBEs, (B_5) y (B_6). Llamamos al algoritmo resultante numeración de valor superlocal (nvs).

Listing 20: Numeración del valor superlocal. Código original.

```
1  B0 :    m0 ← a0 + b0
2          n0 ← a0 + b0
3          (a0 > b0) → B1 , B2
4
5  B1 :    p0 ← c0 + d0
6          r0 ← c0 + d0
7          → B6
```

```
8
9   B_2  :    q_0 ← a_0 + b_0
10             r_1 ← c_0 + d_0
11             (a_0 > b_0) → B_3 , B_4
12
13  B_3  :    e_0 ← b_0 + 18
14             s_0 ← a_0 + b_0
15             u_0 ← e_0 + f_0
16             → B_5
17
18  B_4  :    e_1 ← a_0 + 17
19             t_0 ← c_0 + d_0
20             u_1 ← e_1 + f_0
21             → B_5
22
23  B_5  :    e_2 ← (e_0, e_1)
24             u_2 ← (u_0, u_1)
25             v_0 ← a_0 + b_0
26             w_0 ← c_0 + d_0
27             x_0 ← e_2 + f_0
28             → B_6
29
30  B_6  :    r_2 ← (r_0, r_1)
31             y_0 ← a_0 + b_0
32             z_0 ← c_0 + d_0
```

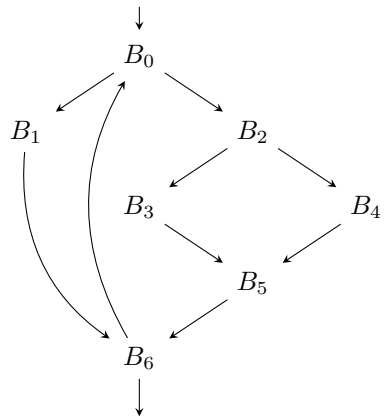


Figura 5: El grafo de flujo de control del listado 20 .

DEFINICIÓN 14 (Función ϕ.) *Una función ϕ toma varios nombres y se fusionan, definiendo un nuevo nombre [5] .*

DEFINICIÓN 15 (Asignación simple estática, ASE.) *Es una representación intermedia que tiene un sistema de nombres basado en el valor, creado por cambio de nombre y uso de pseudo-operaciones llamadas funciones ϕ, ASE codifica tanto el control como el flujo de valores. Es ampliamente utilizado en optimización [5] .*

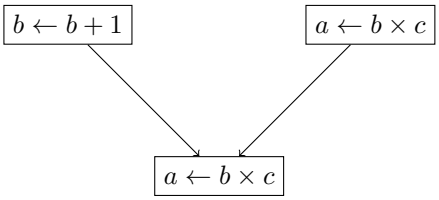
Movimiento de código perezoso

El movimiento de código perezoso (**mcp**) utiliza el análisis de flujo de datos para descubrir ambas operaciones que son candidatos para el movimiento del código y las ubicaciones donde puede colocar las operaciones. El algoritmo opera de forma tal que va del programa y su GFC, en lugar de la forma ASE. El algoritmo utiliza tres conjuntos diferentes de datos ecuaciones de flujo y deriva conjuntos adicionales de esos resultados. Produce, por cada borde en el GFC, un conjunto de expresiones que deberían evaluarse a lo largo de ese arco y, para cada nodo en el GFC, un conjunto de expresiones cuyas evaluaciones expuestas arriba deben eliminarse del bloque correspondiente. Una estrategia de reescritura simple interpreta estos conjuntos y modifica el código.

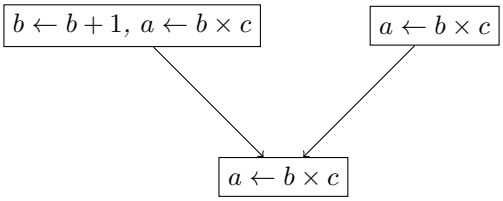
El **mcp** combina el movimiento del código con la eliminación de redundantes y de cálculos parcialmente redundantes. La redundancia se habló en el contexto de la numeración de valores superlocales. Un cálculo es parcialmente redundante en el punto **p** si ocurre en algunos, pero no en todos, los caminos que alcanzan **p** y ninguno de sus operandos constituyentes cambia entre esas evaluaciones y **p**. Los **EMPLOS 6** y **EJEMPLO 7** muestran dos formas de que una expresión puede ser parcialmente redundante. En el **EMPLOS 6**, $a \leftarrow b \times c$ ocurre en una ruta que conduce al punto de fusión, pero no en el otro para que el segundo cálculo sea redundante, las inserciones del **mcp** en una evaluación de una expresión $\leftarrow b \times c$ en el otro camino como se muestra en el **EJEMPLO 6**. En el **EJEMPLO 7**, $a \leftarrow b \times c$ es redundante a lo largo del borde posterior del bucle pero no a lo largo del borde que ingresa al bucle. Insertar una evaluación de un $\leftarrow b \times c$ antes de que el bucle haga que la ocurrencia dentro del bucle sea redundante, como se muestra en el **EJEMPLO 7**. Al hacer que el cálculo invariante de bucle sea redundante y eliminándolo, **mcp** lo mueve fuera del ciclo, una optimización se llama ciclo del movimiento de código invariante cuando se realiza por sí misma [5] .

EJEMPLO 6 *Conversión de redundancia parcial a redundancia:*

Redundancia parcial

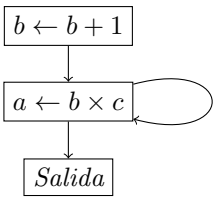


Redundancia

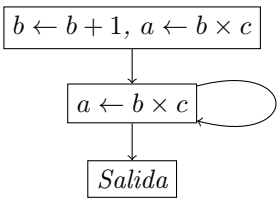


EJEMPLO 7 *Conversión de redundancia parcial a redundancia:*

Redundancia parcial



Redundancia



Ciclos en grafos de flujos

Antes de considerar la optimización de ciclos, hay que definir lo que constituye un ciclo en un grafo de flujo. Se utilizará la noción de que un nodo domina

a otro para definir los ciclos naturales y la clase especial importante de grafos de flujo reducibles [3].

En un grafo de flujo, ¿que es un ciclo, y como se pueden encontrar todos los ciclos? Considerando el GFC de la figura 4 hay un ciclo, formado por el bloque B_2. Un ciclo es una serie de nodos en un grafo de flujo tales que:

1. Todos los nodos en la serie están fuertemente conectados; es decir, desde cualquier nodo dentro del ciclo a cualquier otro, hay un camino de longitud uno o más, siempre dentro del ciclo, y

2. La serie de nodos tiene una entrada única, es decir un nodo dentro del ciclo tal que la única forma de alcanzar un nodo del ciclo desde un nodo fuera del ciclo es atravesar primero la entrada.

Un ciclo que no contiene otros ciclos se denomina ciclo interno [3] .

DEFINICIÓN 16 (Ciclos naturales.) *Dado un arco de retroceso $n \to d$, se define el ciclo natural del arco como d más el conjunto de nodos que puede alcanzar n sin ir a través de d. El nodo d es el encabezamiento del ciclo.*

Una aplicación importante de la información sobre dominadores es determinar los ciclos de un grafo de flujo objeto de mejora. Estos ciclos tienen dos propiedades importantes:

1. *Un ciclo debe tener un solo punto de entrada, llamado "encabezamiento". Este punto de entrada domina todos los nodos dentro del ciclo, o no sería la única entrada al ciclo.*

2. *Debe haber al menos una forma de iterar el ciclo, es decir, al menos un camino de regreso al encabezamiento.*

Una buena forma de encontrar todos los ciclos en un grafo de flujo es buscar arcos en el grafo de flujo cuyas cabezas dominen sus colas. (Si $a \to b$ es un arco, b es la cabeza y a la cola). Dichos arcos se denominan arcos de retroceso [3] .

ALGORITMO 2 (Construcción de un ciclo natural.)
Entrada *Un grafo de flujo G y una arco de retroceso $n \to d$.*

Salida *El conjunto ciclo que consta de todos los nodos dentro del ciclo natural de $n \to d$.*

Método

Comenzando con el nodo $\boldsymbol{n}$*, se considera cada nodo $m \neq d$ que se sabe que está en el ciclo para asegurarse de que los predecesores de* $\boldsymbol{m}$ *también se colocan en el ciclo. Cada nodo del ciclo, excepto* $\boldsymbol{d}$*, se coloca una vez en la pila, así que sus predecesores serán examinados. Observe que como* $\boldsymbol{d}$ *se coloca en el ciclo al inicio, nunca se examinan sus predecesores, y por tanto se encuentran sólo aquellos nodos que alcanzan* $\boldsymbol{n}$ *sin pasar por* $\boldsymbol{d}$*.*

[3].

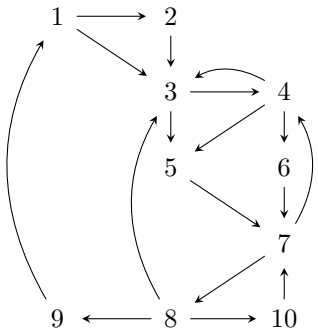


Figura 6: El grafo de flujo con ciclos.

EJEMPLO 8 (Construcción de un ciclo natural.) *De la figura 6 hay un arco $7 \to 4$, y 4 dom 7. De manera similar, $10 \to 7$ es un arco, y 7 dom 10. Los otros arcos con esta propiedad son $4 \to 3$, $8 \to 3$ y $9 \to 1$. Observe que éstas son exactamente los arcos que intuitivamente parecen formar ciclos en el grafo.*

El ciclo natural del arco $10 \to 7$ consta de los nodos 7, 8 y 10, puesto que 8 y 10 son todos los nodos que pueden alcanzar 10 sin pasar por 7. El ciclo natural de $9 \to 1$ es el grafo completo. (No hay que olvidar el camino $10 \to 7 \to 8 \to 9$).
[3].

Grafos de flujo

Para mostrar la transferencia de control entre bloques básicos, usamos un diagrama de flujo. Un grafo de flujo en la Sección 6.1.1; es un grafo dirigido en el que cada nodo es un bloque básico y los arcos muestran las transferencias de control entre los bloques. Cuando se aíslan los bloques básicos, se utilizan las ramas en el programa para determinar sus límites; en el grafo de flujo, estas ramas se usan para unir los bloques. Por ejemplo, el tipo de selección que utilizamos en la Sección 6.1.1 tenía el grafo de flujo mostrado en la Figura 6.1, que se repite aquí por conveniencia: [1]

Análisis basados en regiones

DEFINICIÓN 17 (Intervalo.) *Intuitivamente, un intervalo en un grafo de flujo es un ciclo natural más una estructura acíclica que cuelga de los nodos de un ciclo [3].*

Una propiedad importante de los intervalos es que tienen nodos encabezamiento que dominan todos los nodos en el intervalo; es decir, cada intervalo es una región. Formalmente, dado un grafo de flujo G con nodo inicial n, y un nodo n de G, el intervalo con encabezamiento n, indicado como I(n), se define como sigue:

1. n esta en I(n).

2. Si todos los predecesores de algún nodo $m \neq n_0$ están en I(n), entonces m está en I(n).

3. No hay nada más en I(n).

Por tanto se puede construir I(n) comenzando con n y añadiendo nodos m mediante la regla 2. No importa en qué orden se añadan dos candidatos m porque una vez que los predecesores de un nodo estén todos en I(n), permanecen en I(n), y se añadirá finalmente cada candidato por la regla 2. Llegará un momento, en que no se puedan añadir más nodos a I(n), y el conjunto resultante de nodos será el intervalo con encabezamiento n.

La división de un grafo de flujo en intervalos sirve para imponer una estructura jerárquica en el grafo de flujo. Esta estructura ayuda a su vez a aplicar las reglas para el análisis de flujo de datos dirigido por la sintaxis [3].

ALGORITMO 3 (Análisis de intervalos en un grafo de flujo.)
Entrada. *Un grafo de flujo G con nodo inicial* n_0.

Salida. *Una partición de G en un conjunto de intervalos disjuntos.*

Método.

Para cualquier nodo n, se calcula I(n) por el método esbozado antes.
$I(n) := \{n\}$;
while *existe un nodo* $n \neq m$,
cuyos predecesores están todos en I(n) ***do***
$I(n) := I(n) \cup \{m\}$
Los nodos que sean encabezamientos de intervalos en la partición se eligen de la siguiente manera. Al inicio, ningún nodo está "seleccionado".
constrúyanse $I(n_0)$ *y "seleccionense" todos los nodos en ese intervalo:*
while *hay un nodo m, no "seleccionado" todavía,*
pero con un predecesor seleccionado ***do***
constrúyase I(m) y "seleccionense" todos los nodos en ese intervalo.
[3].

Un intervalo se define como sigue: un intervalo con el encabezado h, denotado como I(h), en una grafo de flujo G = (N, E, i) es un conjunto de nodos de G que se obtiene usando el siguiente algoritmo:

ALGORITMO 4 (Obtención de intervalos.)
Entrada. $I(h) \leftarrow \{h\}$

Salida. *Intervalo I(h).*

Método.

Repetir mientras exista un nodo m en G tal que $m \notin I(h)$ *y* $m \neq i$ *y todos los arcos que ingresan m provienen de nodos en I(h)*

$$I(h) \leftarrow I(h) \cup \{m\}$$

[7].

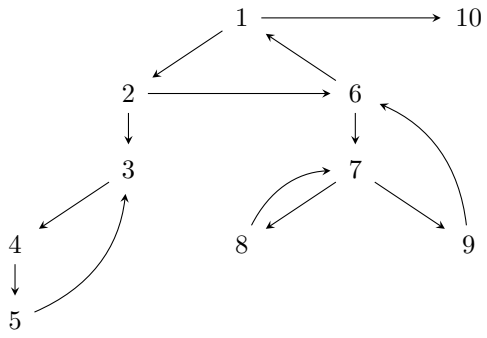


Figura 7: El grafo de flujo con ciclos.

EJEMPLO 9 (Obtención de intervalos.) *Del grafo de flujo de la figura 7 obtener los intervalos, los cuales se presentan a continuación.*

I(1) = {1,2,10}
I(2) = {2}
I(3) = {3,4,5}
I(4) = {4,5}
I(5) = {5}
I(6) = {6}
I(7) = {7,8,9}
I(8) = {8}
I(9) = {9}
I(10) = {10}

Por ejemplo, considere obtener el intervalo con el encabezado 3, es decir, I(3). Primero, I(3) se inicializa a {3}. Luego, el nodo 4 se identifica como un nodo que no está en I(3) cuyos predecesores están todos en I(3) y se suman a I(3). Del mismo modo, solo el nodo 5 el predecesor, el nodo 4, ahora está en I(3), y por lo tanto, I(3) se convierte en {3,4,5}. El nodo 6 no se debe agregar al intervalo porque uno de los predecesores de ese nodo, el nodo 2, no esta en I(3).

Ahora que se han construido todos los mecanismos para realizar el análisis de flujo de datos estructurales, realizar un análisis de intervalo es trivial: es idéntico al análisis estructural, excepto que solo aparecen tres tipos de regiones, a saber, acíclica general, apropiada e impropia. Como ejemplo, considere el grafo de flujo original de la Figura 7.4, que se reproduce en la figura 8.16, junto con su reducción a intervalos. El primer paso gira sobre el bucle que comprende B4 y B6 en el nodo B4a, y el segundo paso reduce la estructura acíclica resultante para el nodo único que entra. El reenvío correspondiente de las funciones de flujo de datos son las siguientes [**?**] :

La propagación de constantes intenta para cada punto de programa v determinar los valores de las variables que tienen cada vez que la ejecución alcanza v. A menudo, una variable tiene valores diferentes en un punto del programa v cuando la ejecución alcanza v varias veces. El análisis de intervalos saca lo mejor de esta situación al calcular un intervalo que encierra todos los valores posibles que la variable puede tener cuando la ejecución llegue a v [**?**] .

DEFINICIÓN 18 (Región.) *Es un conjunto de nodos N que incluye un encabezamiento que domina a los otros nodos en la región. Todas las aristas entre los nodos de N están en la región, excepto algunas de ellas que entran al encabezamiento [3].*

ALGORITMO 5 (Encontrar las regiones en un grafo de flujo.)
Entrada. *El grafo de flujo reducible G = (BLOQUEO, SUC, INITIAL_BLOCK) de un procedimiento.*

Salida. *Las regiones en G.*

Método.

BLOCK, SUC y INITIAL.BLOCK son como en Algorithm CONSTRUCT.FLOW- GRÁFICO dado anteriormente en Sec. 12-6.1. El procedimiento DEPTH usa el vector global MARK y la variable global CTR. DFST es una variable de conjunto global utilizada en el árbol de expansión en profundidad construido por DEPTH. NODE_POSITION da la posición del nodo en una orden de procesamiento de una sola pasada. REGION es un vector de conjuntos que contienen las regiones; r se usa como un índice para este vector. TEMP es un variable temporal utilizada para mantener nodos mientras se están utilizando sus predecesores examinados. STACK es la pila de nodos cuyos predecesores deben ser examinados para la posible inclusión en la región actual. Los subalgoritmos PUSH y POP son asumidos para su uso con esta pila, n es la cantidad de nodos.

1. *-[Inicializar las variables globales para buscar el primer camino del procedimiento]*
   MARK $\leftarrow$ *false*
   DFST $\leftarrow \phi$ *(ϕ denota conjunto vacío)*
   CTR $\leftarrow n$
   $r \leftarrow r_0$
2. *-[Obtener NODE.POSITION y DFST usando la búsqueda del primer camino del procedimiento]*
   llamar DEPTH (INITIAL_BLOCK)
3. *-[Encontrar la región asociada con cada arco de retorno]*
   Repetir para todos los arcos $t \rightarrow h$ *en G dónde*
   NODE_POSITION[t] > NODE_POSITION[h]
   $r \leftarrow r + 1$
   STACK $\leftarrow \phi$
   $REGION_r \leftarrow BLOCK_h$
   If $t \notin REGION_r$ *then* $REGION_r \leftarrow REGION_r \cup BLOCK_t$
   llamar PUSH(STACK,t)
   Repetir que bloque permanece en STACK
   TEMP $\leftarrow$ *POP(STACK)*
   Repetir para cada $p \in PRED[TEMP]$
   If $p \notin REGION_r$
   then $REGION_r \leftarrow REGION_r \cup BLOCK_p$
   Llamar PUSH(STACK, p)
4. *-[Fin]*
   Exit

[7].

Procedimiento DEPTH(i). Este procedimiento recursivo crea un árbol de expansión en profundidad para un grafo que comienza en el nodo **i**. El DFST configurado contiene la expansión en profundidad del árbol. El vector NODE_POSITION da la posición del nodo de acuerdo al orden de la profundidad uno. CTR cuenta desde n, la cantidad de nodos, hasta 1, y se usa para que se asegure de que los nodos se coloquen en NODE_POSITION en el orden inverso al de que fueron visitados por última vez. Todas las variables son globales para el procedimiento.

1. -[Marque el nodo actual como visitado]
   MARK[j] ← true

2. -[Examine los sucesores del nodo actual]
   Repita para cada j ∈ SUC[i]
   if not MARK[j]
   then DFST ← DFSTU(i, j)
   Llamar DEPTH(j)

3. -[Determine la posición del nodo actual en la lista ordenada en profundidad]
   NODE_POSITION[i] ← CTR
   CTR ← CTR - 1

4. -[Fin]
   Return
   [7].

EJEMPLO 10 (Obtención de regiones.) *Se presenta cómo el algoritmo encontraría las regiones en el diagrama de flujo que se muestra en la figura 7 . La llamada inicial a DEPTH con i = 1 hace que MARK[1] sea establecido como* ***true****. Entonces, un sucesor del nodo 1, en este caso uno de los nodos 2 y 10, es elegido. Si se elige el nodo 2, se agrega el arco (1,2) a DFST y una llamada recursiva a DEPTH con i = 2 se hace. Si se supone que en todos los casos, los sucesores examinados en orden de izquierda a derecha, donde los nodos se visitan en el siguiente orden:*

1, 2, 3, 4, 5, 4, 3, 6, 7, 8, 7, 9, 7, 6, 3, 2, 1, 10,1

Al elegir un orden basado en la última visita a cada nodo de derecha a izquierda escaneo de la secuencia anterior, se deriva la secuencia:

5, 4, 8, 9, 7, 6, 3, 2, 10, 1

Invertir este orden da la siguiente secuencia de primera profundidad:

1, 10, 2, 3, 6, 7, 9, 8, 4, 5

Finalmente, podemos derivar las posiciones de los nodos en la lista de la siguiente manera:

Node number 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
NODE-POSITION 1, 3, 4, 9, 10, 5, 6, 8, 7 , 2

En el paso 3, para un borde anterior como 1 → 2, NODE_POSITION [1] = 1 es menos que NODE_POSITION [2] = 3, pero para un borde posterior, como 5 → 3, NODE_POSITION [5] = 10 es mayor o igual a NODE_POSITION [3] = 4. La región correspondiente al borde posterior 5 → 3 se encuentra colocando primero la cabeza, nodo 3, en la región, luego agregar el nodo 5 y luego agregar el predecesor del nodo 5, nodo 4. Las regiones {3,4,5}, {7,8}, {6,7,8,9} y {1,2,3,4,5,6,7,8,9} son encontrado lo correspondiente, respectivamente, a los arcos posteriores 5 → 3, 8 → 7, 9 → 6, y 6 → 1.

Ejercicios

E1.- Considérese el programa para la multiplicación de matrices:

```
1 begin
2     for i := 1 to n do
```

```
3           for j := 1 to n do
4               c[i, j] := 0;
5           for i := 1 to n do
6               for j := 1 to n do
7                   for k := 1 to n do
8                    c[i, j] := c[i, j] + a[i, k] * b[k, j]
9   end
```

1. Genérese código de tres direcciones.
2. Construyase un grafo de flujo a partir de las proposiciones de tres direcciones.
3. Encuéntrense los ciclos en el grafo de flujo.

E2.- Considérese el siguiente programa que cuenta los números primos desde 2 a n utilizando el método de la criba sobre una matriz debidamente grande:

```
1   begin
2       read n;
3       for i := 2 to n do
4           a[i] := true;                    /* inicializacion */
5       cuenta := 0;
6       for i := 2 to n ** .5 do
7           if a[i] then                      /* i es primo */
8               begin
9                   cuenta := cuenta + 1;
10                  for j := 2 * i to n by i do
11                   a[j] := false  /* j es divisible de 2 */
12              end;
13      print cuenta
14  end
```

1. Genérese código de tres direcciones.
2. Construyase un grafo de flujo a partir de las proposiciones de tres direcciones.
3. Encuéntrense los ciclos en el grafo de flujo.

Bibliografía

[1] Thomas W. Parsons, Introduction to compiler construction, Computer Science Press, 1992.

[2] Ralph Grishman, Computational Linguistics An Introduction, Cambridge University Press, 1986.

[3] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compiladores Principios, técnicas y herramientas, Addison Wesley, 1990.

[4] Torben Aegidius Mogensen, Basics of compiler design, DEPARTMENT OF COMPUTER SCIENCE UNIVERSITY OF COPENHAGEN, 2009.

[5] Linda Torczon and Keith D. Cooper, Engineering a Compiler, Second Edition, Elsevier 2012.

[6] Bernard Teufel, Stephanie Schmidt, Thomas Teufel, Compiladores, conceptos fundamentales, Addison-Wesley Iberoamericana, 1995.

[7] Jean Paul Tremblay, Paul G. Sorenson, The theory and practice of compiler writing, McGraw Hill Book Company, 1985.

[8] Mark P. Jones, jacc: just another compiler compiler for Java A Reference Manual and User Guide, Department of Computer Science Engineering OGI School of Science Engineering at OHSU 20000 NW Walker Road, Beaverton, OR 97006, USA February 16, 2004.

[9] www.lcc.uma.es/ galvez/ftp/tci/TutorialYacc.pdf