Libro del dragón Capitulo 1

**Compilador**

Es un programa que lee un programa escrito en lenguaje *fuente,* y lo traduce a un programa equivalente en lenguaje *objeto.*

|  |  |  |
| --- | --- | --- |
| programa  fuente | *compilador* | programa  objeto |
|  | mensajes de error |  |

En la compilación hay dos partes: análisis y síntesis.

La parte del **análisis** divide al programa fuente en sus elementos componentes y crea una representación inter­media del programa fuente.

La parte de la **síntesis** construye el programa objeto deseado a partir de la representación intermedia

análisis lineal o **análisis léxico**

En este el programa fuente se lee de izquierda a derecha y se agrupa en ***componentes léxicos****,* que son secuencias de caracteres que tienen un significado colectivo. La secuencia de caracteres que forma un componente léxico se denomina ***lexema***del componente.

análisis léxico los caracteres de la proposición de asignación

posición := inicial + velocidad \* 60

<id, posición > <:=, > <id, inicial> <+,> <id, velocidad> <\*,> <60,>

se agruparían en los componentes léxicos siguientes:

1. El identificador posición.
2. El símbolo de asignación : =.
3. El identificador inicial.
4. El signo de suma.
5. El identificador velocidad.
6. El signo de multiplicación.
7. El número 60.

**Análisis sintáctico (o jerárquico)**

Este implica agrupar jerárquicamente los com­ponentes léxicos del programa fuente en frases gramaticales (colecciones anidadas) que el compilador uti­liza para sintetizar la salida. Por lo general, las frases gramaticales del programa fuente se representan mediante un árbol de análisis sintáctico

Nota : orden de aparicion de los componentes lexicos

**Análisis semántico**

Ejemplo : El refrigerador verde corría alegremente

Este revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por el análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Un componente importante del análisis semántico es la verificación de tipos. Aquí, el compilador verifica si cada operador tiene operandos permitidos por la es­pecificación del lenguaje fuente.

**Generación de código intermedio**

Después de los análisis sintáctico y semántico, algunos compiladores generan una representación intermedia explícita del programa fuente. Se puede considerar esta representación intermedia como un programa para una máquina abstracta. Esta re­presentación intermedia debe tener dos propiedades importantes; debe ser fácil de producir y fácil de traducir al programa objeto.

*El "código de tres direcciones", que es como el lenguaje ensamblador para una máquina en la que cada posición de memoria puede actuar como un registro.* El código de tres direcciones consiste en una secuencia de instrucciones, cada una de las cuales tiene como máximo tres operandos. El pro­grama fuente de (1.1) puede aparecer en código de tres direcciones como

templ := entareal(60)

temp2 := id3 \* templ

temp3 := id2 + temp2

*idl := temp3*

**Optimación de código**

Este trata de mejorar el código intermedio, de modo que resulte un código de máquina más rápido de ejecutar.

templ := id3 \* 60.0 ,. ..

idl := id2 + templ

**Generación de código**

Este genera el código objeto, que por lo gral. consiste en código de máquina relocalizable o código ensamblador. Las posiciones de memoria se seleccionan para cada una de las variables usadas por el programa. Después, cada una de las instrucciones intermedias se traduce a una secuencia de instrucciones de máquina que ejecuta la misma tarea. Un aspecto decisivo es la asig­nación de variables a registros.

MOVF id3, R2

MULF #60.0, R2

MOVF id2, Rl

ADDF R2, Rl

MOVF Rl, idl

Libro del dragón Capitulo 2

Una *gramática independiente del contexto* tiene cuatro componentes:

1. Un conjunto de componentes léxicos, denominados símbolos *terminales.*

*2.* Un conjunto de no terminales.

3. Un conjunto de producciones, en el que cada producción consta de un no terminal, llamado *lado izquierdo* de la producción, una flecha y una secuencia de componentes léxicos y no terminales, o ambos, llamado *lado derecho* de la producción.

*4. La denominación de uno de los no terminales como símbolo inicial.*

Ejemplo de **GLC**

|  |  |
| --- | --- |
| *lista →* | *lista + dígito* |
| *lista →* | *lista - dígito* |
| *lista →* | *dígito* |
| *dígito →* | *0 | 1 | 2 | 3 |4 | 5 | 6 | 7 |8 | 9* |

o

*lista → lista + dígito* | *lista* - *dígito* | *dígito*

**Arbol de análisis sintáctico**

Indica gráficamente cómo del símbolo inicial de una gramática deriva una cadena del lenguaje.

Formalmente, dada una gramática independiente del contexto, un ***árbol de análisis sintáctico***es un árbol con las propiedades siguientes:

1. La raíz está etiquetada con el símbolo inicial.
2. Cada hoja está etiquetada con un componente léxico o con **ε**.
3. Cada nodo interior está etiquetado con un no terminal.
4. Si *A* es el no terminal que etiqueta a algún nodo interior y *X1 X2 • • Xn, son las etiquetas de los hijos*

*de ese nodo, de izquierda a derecha, entonces A → X1 X2 • • Xn, es una producción. Aquí, X1 X2 • • Xn*

*representa un sím­bolo que es un terminal o un no terminal. Como caso especial, si .A →* **ε***. entonces*

*un nodo etiquetado con A tiene sólo un hijo etiquetado con* **ε*.***

**Ambigüedad**

*Para demostrar que una gramática es ambigua, lo único que se requiere es encontrar una cadena de componentes léxicos que tenga más de un árbol de análisis sintáctico.*

*cadena → cadena + cadena | cadena - cadena |0|l|2|3|4|5|6|7 | 8|9*

**Asociatividad de operadores**

**Definicion dirigidas por la sintaxis**

*Asocia a cada símbolo de la gramática un conjunto de atributos y a cada producción, un conjunto de reglas semánticas para calcular los valores de los atributos asociados con los símbolos que aparecen en esa producción.*

Una traducción es una transformación de una entrada en una salida. La salida para cada entrada *x* se especifica de la forma siguiente. Primero, se construye un árbol de análisis sintáctico para *x.* Supóngase que un nodo *n* del árbol de análisis sintáctico está etiquetado con el símbolo *X* de la gramática. Se escribe *X.a* para indicar el valor del atributo a de X en ese nodo. El valor de *X.a* en *n* se calcula por fa regla semántica para el atributo *a* asociado con la producción de *X* utilizada en el nodo *n. Al árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se dice que es un árbol de análisis sintáctico con anotaciones.*

**Atributos sintetizados**

Se dice que un atributo está sintetizado si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de atributos de los hijos del nodo.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Producción | | | Regla Semántica | |
| *expr →* *expr1 +término*  *térmi* | | | *expr := expr1.t +término.t*  *= expr¡.t término.t '+'* | |
| *expr → expr1 - término* | | | *expr := expr1.t – término.* | |
| *expr →* *termino* | | | *expr.t* | = *término.t* |
| *término →* 0 |  | *término,t* ;**='0'**  **= '0'** | | |
| *término →* 1 |  | *término.t* :**='1'**  **= '1'** | | |
| *término →* 9 |  | *término. t :****=* '9'** | | |

|  |  |  |
| --- | --- | --- |
| Producción | Regla Semántica | |
| *sec →* **comienza** | *sec.x*  *sec. y* | = 0 = 0 |
| *sec →* *sec instr* | *sec.x sec. y* | = *sec.x + instr.dx = sec.y + instr.dy* |
| *instr →* **este** | *instr.dx instr.dy* | = 1 = 0 |
| *instr →* **norte** | *instr.dx instr. dy* | = 0  = 1 |
| *instr →* **oeste** | *instr.dx i instr.dy* | = -1 = 0 |
| Instr → **sur** | *instr.dx instr.dy* | = 0  = -1 |

**Recorridos en profundidad**

***void*** *visita (Nodo n*){

**for** (cada hijo *m* de *n,* de izquierda a derecha )

*visita (ni);*

evalúa reglas semánticas en el nodo *n*

}

Se llama "en profundidad" porque siempre que pueda, visita a un hijo no vi­sitado de un nodo, de modo que intenta visitar los nodos más alejados de la raíz lo antes posible.

**Esquema de traducción**

Es una GLC en la que se encuentran intercalados, en los lados derechos de las producciones, fragmentos de programa llamados *acciones semánticas. E*s como una definición dirigida por la sintaxis, con la excep­ción de que el orden de evaluación de las reglas semánticas se muestra explícita­mente.

*expr → expr* + *término { print* ('+') }

*expr →* *expr* + *término*  { *print {'-') }*

*expr →* *término*

*término →* 0 { *print* ('0') }

*término →* 1 { *print ('1')}*

*...*

*término →*  9 { *print* ('9') }

**Análisis sintáctico descendente**

En este la construcción se inicia en la raíz y avanza hacia las hojas

1. En el nodo *n,* etiquetado con el no terminal *A,* selecciónese una de las produc­ciones para *A* y construyase los hijos de *n* para los símbolos del lado derecho de  
   la producción.
2. Encuéntrese el siguiente nodo en el que ha de construirse un subárbol.

*tipo →* *simple*

| ^ id

| **array** [ *simple ]* of *tipo  
simple →* **integer** (2.8)

**| char**

**| núm puntopunto núm**

**Análisis sintáctico predictivo**

El *análisis sintáctico descendente recursivo* es un método descendente en el que se ejecuta un conjunto de procedimientos recursivos para procesar la entrada. A cada no terminal de una gramática se asocia un procedimiento.

Un *analizador sintáctico predictivo* es un programa que consiste en un procedi­miento para cada no terminal.

**análisis sintác­tico predictivo**, en el que el símbolo de preanálisis determina sin ambigüedad el pro­cedimiento seleccionado para cada no terminal. La secuencia de procedimientos lla­mados en el procesamiento de la entrada define implícitamente un árbol de análisis sintáctico para la entrada.

**void** *parea* ***(complex t*) {**

**if (***preanálisis* == *t* ***)***

*preanálisis= sigcomplex();*

**else** *error();*

**}**

**void** *tipo()* ***{***

**if (***preanálisis==* ***INTEGER* ||** *preanálisis==* **CHAR** || *preanálisis ==* **NUM** ) {

*simple();*

**else if (***preanálisis* = '^' **)**{

*parea* ('^'); *parea* **(id);**

**}**

**else if (***preanálisis ==* ***ARRAY* ) {**

*parea* **(*ARRAY* );** *parea* ('['); *simple(); parea* (']'); parea **(OF);** tipo();

**}**

**else** error()**;**

**}**

**void** *simple()* ***{***

**if (***preanálisis* == **INTEGER)**

*parea* **(INTEGER ) ;**

**else if (***preanálisis* == **CHAR)**

*parea* **(CHAR);**

**else if (***preanálisis* == **NUM ){**

*parea* **(NUM);** *parea* **(PUNTOPUNTO );** *parea* **(NUM)**

**}**

**else** *error()* **;**

**}**

Seudocódigo de **un analizador sintáctico predictivo.**

El análisis sintáctico predictivo depende de la información sobre los primeros símbolos que pueden ser generados por el lado derecho de una producción. Para precisar, sea a el lado derecho de una producción para el no terminal *A.* Se define PRIMERO(a) como el conjunto de componentes léxicos que opere como los primeros símbolos de una o más cadenas generadas a partir de a. Si a es **c** o puede generar e, entonces e también está en PRIMERO(a)2. Por ejemplo,

PRIMERO *(simple) =* {**integer, char, núm** }

PRIMERO (^ **id)** = {**^**}

PRIMERO **(array** [ *simple* ] of *tipo) =* { **array** }

En la práctica, muchos lados derechos de una producción comienzan con compo­nentes léxicos, lo que simplifica la construcción de conjuntos PRIMERO.

Se deben considerar los conjuntos PRIMERO si hay dos producciones *A-> alfa A* -> beta*.* El análisis sintáctico descendente recursivo sin retroceso requiere que PRIMERO(alfa) y PRIMERO(beta) sean disjuntos. El símbolo de preanálisis se puede usar entonces para decidir el tipo de producción a utilizar; si el símbolo de preanálisis está en PRIMERO(alfa), entonces se usa alfa. De otro modo, si el símbolo de pre­análisis está en PRIMERO(beta), entonces se usa beta.

**Cuándo se usan las producciones ε**

El ana­lizador sintáctico descendente recursivo usará una producción epsilon por defecto cuando no se pueda aplicar otra producción. Por ejemplo, considérese:

*prop* → begin *props\_opc* end

*props-opc →* *lista\_props* | **ε**

Durante el análisis sintáctico de *props\_opc,* si el símbolo de preanálisis no está en *PRlMERO(lista-props),* entonces se usa la producción epsilon. Esta elección es justo la correcta si el símbolo de preanálisis es **end.** Cualquier símbolo de preanálisis di­ferente de **end** dará como resultado un error, que se detectará durante el análisis sin­táctico *de prop.*

**Diseño de un analizador sintáctico predictivo**

Cada procedimiento hace dos cosas:

1. Decide la producción que utilizará analizando el símbolo de preanálisis. Si el símbolo de preanálisis está en PRIMERO(a), se usa la producción con lado derecho a. Si hay un conflicto entre dos lados derechos de cualquier símbolo de preanálisis, entonces en esa gramática no se puede emplear este método de aná-lisis sintáctico. Si el símbolo de preanálisis no está en el conjunto PRIMERO de ningún otro lado derecho, se usa una producción con € en el lado derecho.

2. El procedimiento usa una producción imitando al lado derecho. Un no terminal da como resultado una llamada al procedimiento del no terminal, y un compo­nente léxico que coincida con el símbolo de preanálisis da como resultado que se lea el siguiente componente léxico. Si el componente léxico de la producción no coincide en algún punto con el símbolo de preanálisis, se declara un error. La figura 2.17 es el resultado de aplicar estas reglas a la gramática (2.8).

Al igual que se crea un esquema de traducción extendiendo una gramática, se puede crear un traductor dirigido por la sintaxis extendiendo un analizador sintác­tico predictivo. En la sección 5.5 se da un algoritmo para este propósito. La si­guiente construcción limitada es suficiente por el momento, pues los esquemas de traducción que se implantan en este capítulo no asocian atributos a los no termi­nales:

1. Construyase un analizador sintáctico predictivo, ignorando las acciones en las  
   producciones.
2. Copíense las acciones del esquema de traducción en el analizador sintáctico. Si  
   una acción aparece después del símbolo gramatical *X* en la producción *p,* entonces se copia después del código que implanta *X.* De otro modo, si aparece al prin­cipio de la producción, entonces se copia justo antes del código que implanta la producción. " ,

En la siguiente sección se construirá un traductor de este tipo.

**Recursividad por la izquierda**

Es posible que un analizador sintáctico descendente recursivo entre en un lazo (bu­cle) indefinido.

Hay un problema con producciones recursivas por la izquierda del tipo

*expr →* *expr + término*

en la que el símbolo situado más a la izquierda del lado derecho es el mismo que el no terminal del lado izquierdo de la producción.

Para eliminar la recursividad por la izquierda las producciones

A → A | 

se transforman en

A → 

R → R | **ε**

Libro del dragón Capitulo 3

**AUTOMATA FINITO**

**Autómata finito no determinista**

Un autómata finito no determinista NFA consiste en :

1. Un conjunto de estados finito S .

2. Un conjunto de símbolos de entrada sigma ( el alfabeto de entrada ).

3. Una función de transición mueve que transforma pares estado-símbolo en

conjuntos de estados.

4. Un estado de s0 que se considera el estado de inicio (o inicial) .

5. Un conjunto de estados F considerados como estados de aceptación (o finales).

|  |  |  |
| --- | --- | --- |
| estado | símbolos de entrada | |
| a | b |
| 0 | { 0 , 1 } | { 0 } |
| 1 |  | { 2 } |
| 2 |  | { 3 } |
| 3 |  |  |

tabla de transición del NFA

**Autómatas finitos deterministas**

Un autómata finito determinista DFA es un caso especial de un NFA donde

1. ningún estado tiene una transición epsilon, es decir una transición con la entrada epsilon.

2. Para cada estado s y cada símbolo de entrada a, hay a lo sumo una arista etiquetada a que sale de s.

**Simulación de un DFA .**

ENTRADA : Una cadena de entrada x que termina con un carácter de fin de archivo. Un DFA D con el estado de inicio s0, y un conjunto de estados de aceptación F y una función de transición.

SALIDA : respuesta “sí”, si D acepta x, “no" de otro modo .

MÉTODO : Aplicar el algoritmo de la figura . 3,27 para la cadena de entrada x . La función mueve(s,c) da el estado al cual hay una arista desde el estado s en la entrada c . La función sigtecar devuelve el siguiente carácter de la cadena de entrada x .

s = s0 ;

c = sigtecar();

while (c != EOF) {

s = mueve(s, c); c = sigtecar() ;

}

if (s is in F)

return "yes" ;

else return "no" ;

Algoritmo 2: Construcción de subconjuntos. Construcción de un AFD a partir de un AFN.

Entrada. Un AFN N.

Salida. Un AFD D que acepta el mismo lenguaje.

Método. El algoritmo construye una tabla de transiciones tranD para D. Cada estado del AFD es un conjunto de estados del AFN y se construye tranD de modo que D simulará “en paralelo” todos los posibles movimientos que N puede realizar con una determinada cadena de entrada.

Operaciones sobre los estados de un AFN

|  |  |
| --- | --- |
| OPERACION | DESCRIPCIÓN |
| cerradura- epsilon (s) | Conjunto de estados del AFN alcanzables desde el estado s del AFN con transiciones epsilon solamente |
| cerradura- epsilon (T) | Conjunto de estados del AFN alcanzables desde algún estado s en T con transiciones epsilon solamente. |
| mover(T, a) | Conjunto de estados del AFN hacia los cuales hay una transición con el símbolo de entrada a desde algún estado s en T del AFN. |

al inicio cerradura- epsilon(s0) es el único estado dentro de estadosD y no esta marcado;

while (haya un estado no marcado T en estadosD){

marcar T;

for(cada símbolo de entrada a){

U:= cerradura-epsilon(mueve(T, a));

if U no esta en estadosD then

añadir U como estado no marcado a estadosD;

tranD(T, a):= U

}

}

Figura 4. La construcción de subconjuntos

meter todos estados de T en pila;

inicializar cerradura-epsilon(T) a T;

while (pila no este vacía) {

sacar t , el elemento tope de pila ;

for(cada estado u con una arista desde t a u etiquetada con epsilon)

if (u no esta en cerradura-epsilon (T)){

añadir u a cerradura-epsilon(T) ;

meter u en pila ;

}

}

Calculando la cerradura-epsilon

Algoritmo:construcción de un AFN a partir de una expresión regular

Entrada: Una expresión regular r en un alfabeto S

Salida: Un AFN N que acepte L(r)

Para vacío, construir el AFN

Para a del alfabeto, construir el AFN

Supóngase que N(s) y N(t) son AFN para las expresiones regulares s y t

Para las expresiones regular s | t, constrúyase el siguiente AFN compuesto N(s | t )

Para la expresión regular st, constrúyase el AFN, compuesto N(st)

Para la expresión regular s\*, constrúyase el AFN compuesto N(s\*)

Para la expresión regular entre paréntesis (s), utilícese N(s) como AFN.

Cada vez que se construye un nuevo estado, se le da un nombre distinto. De esta forma, no puede haber dos estados de un AFN componentes con el mismo nombre. Aunque aparezcan los mismos símbolos varias veces en r, se crea para cada ejemplo ese símbolo un AFN aparte con sus propios estados.

Algoritmo 3.22 : Simulación de un AFN .

ENTRADA : Una cadena de entrada x terminada por un carácter de final de archivo eof. un AFN N con un estado de inicio s0 , y un conjunto de estados de aceptancion F.

SALIDA : respuesta “sí" si N acepta x ; “no" de otro modo .

MÉTODO :

S := cerradura-epsilon({s0});

a = sigtecar();

while( a != eof ) {

S := cerradura-epsilon(mueve(S,a));

a = sigtecar();

}

if ( S intserseccion F != vacio ) return "si" ;

else return "no" ;

**Figure 3.37: Simulando un AFN**

Libro del dragón Capitulo 4

**Notación**

1. Estos símbolos son **terminales** :

a. las primeras letras minúsculas del alfabeto , como a, b, c.

b. símbolos de operador como + , - y así sucesivamente.

c. símbolos de puntuación como paréntesis , comas , y así sucesivamente.

d. Las dígitos 0 , 1 , …, 9 .

e. Cadenas en negrita , tales como **id** o **if** , cada uno de los cuales representa una solo símbolo terminal .

2. Estos símbolos son **no terminales** :

a. las letras mayúsculas al principio del alfabeto, como A, B , C .

b. La letra S , que , cuando aparece, suele ser el símbolo de inicio .

c. minúscula , cursiva nombres tales como expr o prop .

d. Cuando se habla de construcciones de programación , las letras mayúsculas puede ser se utiliza para representar no terminales para las construcciones . Por ejemplo , no terminales para expresiones, términos y factores a menudo son representados por E , T, y F , respectivamente.

3. Las ultimas letras mayúsculas del alfabeto , tales como X , Y, Z , representan **símbolos de la gramática**; es decir, o terminales o no terminales .

4. Las ultimas letras minúsculas del alfabeto , principalmente u , v , ... z, representan **cadenas de terminales** .

5. letras griegas minúsculas alfa ,beta ,gamma por ejemplo, representan **cadenas de símbolos gramaticales** . Por lo tanto, una producción genérica se puede escribir

A→ .

6. Un conjunto de producciones de A→ 1, A → 2, ..., A→ k con una cabeza común Un llamarlos A - producciones , puede ser escrita A→ 1| 2, ...,| k

7.  A menos que se indique lo contrario , la cabeza de la producción en primer lugar es el símbolo inicial .

Deriva en un paso →

Deriva en cero o mas pasos → ( \* )

Deriva en uno o mas pasos → ( + )

L(G) solo símbolos terminales. W esta en L(G) si y solo si S→w. w es frase de G.

**Gramáticas equivalentes** son aquellas que generan el mismo lenguaje.

Forma de frase S → 

En una derivación por la izquierda se sustituye en cada paso el no terminal mas a la izquierda de cualquier forma de frase

En una derivación por la derecha se sustituye en cada paso el no terminal mas a la derecha de cualquier forma de frase

Si alfa deriva a beta por la derecha se dice que alfa es una **forma de frase derecha**.

Se puede eliminar la recursividad explicita manteniendo una pila.

a

+

b

$

ENTRADA

Programa para análisis sintáctico predictivo

*X*

PILA

SALIDA

Tabla de análisis sintáctico M

*Y*

*Z*

$

Algoritmo de Análisis Sintáctico Predictivo no recursivo

Al principio la pila contiene el símbolo inicial de la gramática encima de $.

apuntar ae al primer símbolo de w$

repetir

X  símbolo de la cima de la pila

a  símbolo apuntado por ae

si x es un terminal o $ entonces

si X = a entonces

extraer X de la pila

avanzar ae

si\_no error fin\_si

si\_no

si M[x,a] = X → Y1Y2...Yk entonces

extraer X de la pila

meter Yk, Yk-1, ..., Y1 en la pila, con Y1 en la cima

emitir la producción X → Y1Y2...Yk

si\_no error fin\_si

fin\_si

hasta\_que X = $ /\**la pila está vacia*\*/



**Gramática**

*E* → *TE’*

*E’* → **+***TE’* | ****

*T* → *FT’*

*T’* → **\****FT’* | ****

*F* → **(***E***)** | **id**

**tabla de análisis sintáctico**



|  |  |  |  |
| --- | --- | --- | --- |
| **Gramática ambigua** |  | *PRIMERO*(): | *SIGUIENTE* (*A*) |
| *P* → *iEtPP’* | a  *P’* → *eP* | ****  *E* → *b* |  | Primero(P) = {i, a}  Primero(P') = {e , ε }  Primero(E) = { b} | Siguiente(P) = Siguiente(P') = { $, e }  Siguiente(E) = { t } |

**tabla de análisis sintáctico**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| No terminal | Símbolo de entrada | | | | | |
| a | b | e | i | t | $ |
| P | P→a |  |  | P→iEtPP' |  |  |
| P' |  |  | P'→eP  P'→ **** |  |  | P'→  **** |
| E |  | E→b |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| Pila | Entrada | Acción |
| $P | if b t if b t a e a$ | P -> if E t P P' |
| $P' P t E i | if b t if b t a e a$ | match |
| $P' P t E | b t if b t a e a$ | E->b |
| $P' P t b | b t if b t a e a$ | match |
| $P' P t | t if b t a e a$ | match |
| $P' P | if b t a e a$ | P -> if E t P P' |
| $P' P' P t E i | if b t a e a$ | match |
| $P' P' P t E | b t a e a$ | E->b |
| $P' P' P t b | b t a e a$ | match |
| $P' P' P t | t a e a$ | match |
| $P' P' P | a e a$ | P->a |
| $P' P' a | a e a$ | match |
| $P' P' | e a$ | P'→eP (elimina ambigüedad con regla del else mas próximo) |
| $P' P e | e a$ | match |
| $P' P | a$ | P->a |
| $P' a | a$ | match |
| $P' | $ | P'→ **** |
| $ | $ | aceotar |

La cadena es válida y se acepta pues en la pila y en la entrada sólo queda el símbolo $

**tabla de análisis sintáctico**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Gramática** |  |  | ( | ) | ; | x | $ |
| S → ( A )  A → CB  B → ; A | ε  C → x | S |  | S | S →(A) |  |  |  |  |
|  | A | A →CB |  |  | A →CB |  |
|  | B |  | B→ ε | B →;A |  |  |
|  | C | C →S |  |  | C → x |  |

|  |  |  |
| --- | --- | --- |
| pila | entrada | producción usada (salida) |
| $ S  $ )A(  $ )A  $ )BC  $ )Bx  $)B  $)A;  $)A  $)BC  $)BS  $)B)A(  $)B)A  $)B)BC  $)B)Bx  $)B)B  $)B)  $)B  $)  $ | (x;(x))$  (x;(x))$  x;(x))$  x;(x))$  x;(x))$  ;(x))$  ;(x))$  (x))$  (x))$  (x))$  (x))$  x))$  x))$  x))$  ))$  ))$  )$  )$  $ | S → (A)  A → CB  C → x  B → ;A  A → CB  C → S  S → (A)  A → CB  C → x  B → ε  B → ε |

***PRIMERO* y *SIGUIENTE***

Se facilita la construcción de un analizador sintáctico predictivo con dos funciones asociadas a la gramática G:

***PRIMERO*()** Conjunto de terminales que inician cadenas derivadas de . Si  \* , entonces  también está en primero.

***SIGUIENTE* (*A*)** Conjunto de terminales *a* que pueden aparecer inmediatamente a la derecha de A. *S* \* *Aa* para algún  y . Entre *A* y *a* pudieron existir derivaciones que terminaron en . Es decir si *S* \* *ABCa*, y *B* \*  y *C* \* , entonces *a* esta en *SIGUIENTE* (*A*). Si A puede ser el símbolo situado mas a la derecha en una forma de frase entonces $ esta en *SIGUIENTE* (*A*)

Para calcular todos los *PRIMERO* (*X*)

1. Si *X* es terminal, entonces *PRIMERO* (*X*) es {X}

2. Si *X* → , es una producción entonces añádase  a *PRIMERO* (*X*)

3. Si *X* es no terminal y *X* → *Y1*, *Y2*, .. *Yk*, entonces poner *a* en *PRIMERO* (*X*) si, para alguna *i*, *a* está en *PRIMERO* (*Yi*) y *Y1*, *Y2*, .. *Yi-1* \* .

Para calcular *SIGUIENTE* (*A*)

1. Poner $ en *SIGUIENTE* (*S*), donde *S* es el símbolo inicial

2. Si hay una producción *A* → *B*, entonces todo lo que esté en *PRIMERO* (**) excepto  se pone en *SIGUIENTE* (*B*)

3. Si hay una producción *A* → *B* o una producción *A* → *B*, donde *PRIMERO* (**) contenga , entonces todo lo que esté en *SIGUIENTE* (*A*) se pone en *SIGUIENTE* (*B*)

**Ejemplo**

|  |  |
| --- | --- |
| *PRIMERO*(): | *SIGUIENTE* (*A*) |
| *PRIMERO* (*E*) = *PRIMERO* (*T*) = *PRIMERO* (*F*) = {**(**, **id**}  *PRIMERO* (*E’*) = {**+**, ****}  *PRIMERO* (*T*’) = {**\***, ****} | *SIGUIENTE* (*E*) = *SIGUIENTE* (*E’*) = *P* {**)**, **$**}  *SIGUIENTE* (*T*) = *SIGUIENTE* (*T*’) = {**+**, ), **$**}  *SIGUIENTE* (*F*) = {**+**, **\***, ), **$**} |

**Construcción de tablas análisis sintáctico predictivo no recursivo**

**Construcción de tablas de análisis sintáctico**

1. Para cada producción *A* →  de la gramática, dense los pasos 2 y 3

2. Para cada terminal a de *PRIMERO* (), añádase *A* →  a M[*A*, *a*]

3. Si  está en *PRIMERO* (), añádase *A* →  a M[*A*, *b*] para cada terminal *b* de SIGUIENTE (*A*). Si  está en *PRIMERO* () y $ esta en SIGUIENTE (*A*), añádase *A* →  a M [*A*, $]

4. Hacer cada entrada no definida de M sea error.

**MANGO (handle)**

Un **mango** de una forma de frase derecha g es una producción S →  y una posición de g donde la cadena  podría encontrarse y sustituirse por A para producir la forma de frase derecha previa en una derivación por la derecha de g.

PREFIJOS VIABLES

Este método de análisis garantiza que el mango siempre aparecerá en la cima de la pila, nunca dentro

Los prefijos de las formas de frase derecha que pueden aparecer en la pila se denominan *prefijos viables*

Un **prefijo viable** es un prefijo de una forma de frase derecha que no continua más allá del extremo derecho del mango situado más a la derecha de esta forma de frase.

PROGRAMA PARA ANÁLISIS SINTACTICO LR

apuntar *ae* al primer símbolo de *w*$

repetir

sea S el estado en la cima de la pila y

*a* el símbolo apuntado por *ae*

si *accion* [S,a]= desplazar S’ entonces

meter *a* y después S’ en la cima de la pila

avanzar *ae* al siguiente símbolo de entrada

si\_no

si *acción* [S, a] = reducir A → β entonces

sacar 2 \* |β| símbolos de la pila

sea S’ el estado que hay ahora en la cima de la pila

meter en la cima de la pila A y después *ir\_a* [S’, A]

emitir la producción A → β

si\_no

si acción [S, a] = aceptar entonces

fin con éxito

si\_no error

fin\_repetir

|  |  |
| --- | --- |
| Gramática | Códigos de las acciones |
| (1) E → E + T  (2) E → T  (3) T → T \* F  (4) T → F  (5) F → (E)  (6) F → id | di significa desplazar y meter en la pila el estado i  rj significa reducir por la regla de número j  Acep significa aceptar  El espacio en blanco significa error |

TABLA DE ANÁLISIS SINTÁCTICO LR

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Estado | acción |  |  |  |  |  | ir\_a |  |  |
|  | id | + | \* | ( | ) | $ | E | T | F |
| 0 | d5 |  |  | d4 |  |  | 1 | 2 | 3 |
| 1 |  | d6 |  |  |  | acep |  |  |  |
| 2 |  | r2 | d7 |  | r2 | r2 |  |  |  |
| 3 |  | r4 | r4 |  | r4 | r4 |  |  |  |
| 4 | d5 |  |  | d4 |  |  | 8 | 2 | 3 |
| 5 |  | r6 | r6 |  | r6 | r6 |  |  |  |
| 6 | d5 |  |  | d4 |  |  |  | 9 | 3 |
| 7 | d5 |  |  | d4 |  |  |  |  | 10 |
| 8 |  | d6 |  |  | d11 |  |  |  |  |
| 9 |  | r1 | d7 |  | r1 | r1 |  |  |  |
| 10 |  | r3 | r3 |  | r3 | r3 |  |  |  |
| 11 |  | r5 | r5 |  | r5 | r5 |  |  |  |

MOVIMIENTOS DEL ANALIZADOR SINTÁCTICO LR (para la entrada id \* id + id )

|  |  |  |  |
| --- | --- | --- | --- |
| PILA |  | ENTRADA | ACCION |
| (1) | 0 | id \* id + id $ | desplazar |
| (2) | 0 id 5 | \* id + id $ | reducir por F → id |
| (3) | 0 F 3 | \* id + id $ | reducir por T → F |
| (4) | 0 T 2 | \* id + id $ | desplazar |
| (5) | 0 T 2 \* 7 | id + id $ | desplazar |
| (6) | 0 T 2 \* 7 id 5 | + id $ | reducir por F → id |
| (7) | 0 T 2 \* 7 F 10 | + id $ | reducir por T → T \* F |
| (8) | 0 T 2 | + id $ | reducir por E → T |
| (9) | 0 E 1 | + id $ | desplazar |
| (10) | 0 E 1 + 6 | id $ | desplazar |
| (11) | 0 E 1 + 6 id 5 | $ | reducir por F → id |
| (12) | 0 E 1 + 6 F 3 | $ | reducir por T → F |
| (13) | 0 E 1 + 6 T 9 | $ | reducir por E → E + T |
| (14) | 0 E 1 | $ | aceptar |

Un **elemento LR (0)** de una gramática G es una producción de G con un punto en cierta posición del lado derecho.

Por ejemplo la producción A→XYZ produce los siguientes elementos:

A →.XYZ

A →X.YZ

A→ XY.Z

A →XYZ.

La producción A→ ε genera solo un elemento A→.

1. Inicialmente, agregar todo elemento en I a cerradura (I).

2. If A→ α .B β esta en cerradura (I) and B→ es una producion, entonces agregar el elemento B→. a cerradura (I), si todavia no esta ahi. Aplicar esta regla hasta que no mas elementos nuevos puedan ser agregados a cerradura (I).

void cerradura(I) {

J=I

repeat

for (cada elemento A → α .B β en J)

for (cada producción B →. γ de G)

if (B →. γ no está en J)

agregar B →. γ a J

until no se agreguen mas elementos a J en una ronda

return J

}

**elementos nucleares**

incluyen el elemento inicial S’→ .S y todos los elementos cuyos puntos no estén en el extremo izquierdo

**elementos nucleares**

cuyos puntos estan en el extremo izquierdo

Función ir\_\_A(I,X), donde I es un conjunto de elementos y X es un símbolo gramatical.

**ir\_\_a**

ir\_\_a (I,X) se define como la cerradura del conjunto de todos los elementos A →α X. β , de tal forma que A→ α .X β se encuentre en I.

void elementos(G’) {

C= {cerradura( {[S’→ .S]})}

repeat

for (cada conjunto de elementos I en C y cada símbolo gramatical X tal que ir\_\_A(I,X) no este

vacio y no este en C)

agregar ir\_\_A(I,X) a C

until no se puedan añadir mas conjuntos de elementos a C

return J

}

**Algoritmo: Construcción de una tabla de análisis sintáctico SLR**

**Entrada** : Una gramática aumentada G’

**Salida** : Las funciones ACCION e ir\_a de la tabla de análisis sintáctico SLR para G’ .

**Método** :

1.-Construir C={I0, I1, …, In}, la colección de conjuntos de elementos LR(0) para G’.

2.-El estado i se construye a partir de Ii. Las acciones de análisis sintáctico para el estado i se determinan de la siguiente forma:

a) Si [A→ α .a β ] está en Ii e ir\_\_A(Ii,a)=Ij, entonces establecer ACCION[i.a] a “desplazar j”. Aquí, a debe ser un terminal.

b) Si [A→ α .] esta en Ii, entonces establecer ACCION[i,a] a “reducir A→ α ” para toda a en SIGUIENTE(A); aquí, A tal vez no sea S’.

c) Si [S’→ S.] esta en Ii, entonces establecer ACCION[i,$] a “aceptar”.

Si resulta cualquier acción conflictiva debido a las reglas anteriores, decimos que la gramática no es SLR(1). El algoritmo no produce un analizador sintáctico en este caso.

3. Las transiciones de ir\_\_A para el estado i se construyen para todos lo no terminales A usando la regla: Si ir\_\_A(Ii, A)=Ij, entonces ir\_\_A[i, A]=j.

4. Todas las entradas que no esten definidas por las reglas (2) y (3) se dejan como “error”.

5. El estado inicial del analizador sintáctico es el que se construyó a partir del conjunto de elementos que contienen [S’→ .S].

**elemento LR (1)**

[A→ α . β , a] donde A→ αβ es una producción y a es un terminal o $

[A→ α . , a] pide reducir por A→ α solo si el siguiente símbolo de entrada es a

**elemento LR (1) valido**

Formalmente, se dice que un **elemento LR(1**) [A→ α . β, a] es **válido** para un prefijo viable δα si existe una derivacion S→ δAw → δαβw donde a es 1er simbolo de w , o w es **ε** y a es $

funcion cerradura (I) {

repeat

for cada elemento [A→ α **.**B β, a ] en I, cada producción B → γ de G' y cada b en PRIMERO(βa)

tal que [B →**.** γ , b] no está en I do

agregar [B →**.** γ , b] a I

until no se pueda agregar mas elementos a I

return I

}

void goto(I, X){

sea J el conjunto de items [ A→ α X **.** β, a] tal que [ A→ α **.** X β, a] está en I

return cierre(J)

}

void elementos(G’) {

C= {cerradura( {[S’→ .S, $]})}

repeat

for (cada conjunto de elementos I en C y cada símbolo gramatical X tal que ir\_\_A(I,X) no este

vacio y no este en C)

agregar ir\_\_A(I, X) a C

until no se puedan añadir mas conjuntos de elementos a C

return J

}

**Algoritmo: Construcción de una tabla de análisis sintáctico LR canonico**

**Entrada** : Una gramática aumentada G’

**Salida** : Las funciones ACCION e ir\_a de la tabla de análisis sintáctico **LR canonico** para G’ .

**Método** :

1. Construir C = { I0, I1, ..., In }, la colección de conjuntos de items LR(1) para G’.

2. El estado i se construye a partir de Ii . Las acciones para el estado i se determinan:

a) Si [A → α .a β, b] está en Ii y goto(Ii , a) = Ij , entonces accion[i, a] = desplazar j. a debe ser un terminal.

b) Si [A → α· , a] está en Ii  , entonces accion[i, a] = reducir por A → α, siendo A distinto de S’.

c) Si [S’→ S·] está en I, entonces accion[i, $] = aceptar.

Si se produce algun conflicto al aplicar las reglas anteriores, decimos que la gramática no es LR(1). El algoritmo no puede producir un analizador en este caso.

3. Las transiciones goto para el estado i se construyen usando la regla: if goto(Ii , A) = Ij , then goto[i, A] = j para todos los no terminales A.

4. Todas las entradas no definidas por (2) y (3) serán error.

5. El estado inicial del analizador es el que se construye a partir del conjunto de items que contiene [S’→ ·S, $]

Toda gramática SLR(1) es LR(1), pero para una gramática SLR(1) el analizador LR canónico puede tener más estados que el analizador SLR de esa misma gramática.

Libro del dragón Capitulo 5

**Atributos sintetizados**

Se dice que un atributo está sintetizado si su valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de los atributos de los hijos del nodo.

**Atributos heredados**

Es aquel cuyo valor en un nodo del árbol de análisis sintáctico se determina a partir de los valores de los atributos del padre o de los hermanos del nodo.

D →T L { L.her = T.tipo; }

T →**int** { T.tipo = integer; }

T →**real** { T.tipo = real; }

L →L , **id** { L.her = L.her; añadetipo(id,L.her); }

L →**id** { añadetipo(id, L.her); }

**Grafos de dependencias**

Si un atributo **b** en un nodo de un árbol de análisis sintáctico depende de un atributo **c**, entonces se debe evaluar la regla semántica para **b** en ese nodo después de la regla semántica que define a **c**. las dependencias entre los atributos heredados y sintetizados en los nodos de un árbol de análisis sintáctico se pueden representar de manera muy natural mediante un grafo dirigido llamado grafo de dependencias.

**Construccion del Grafo de dependencias**

Se usa el siguiente algoritmo:

**Nodos**:

for cada nodo n del árbol sintáctico, do

for cada atributo a asociado al símbolo gramatical del nodo n do

construir un nodo etiquetado con a en el grafo de dependencias.

**Arcos**:

for cada nodo n del árbol sintáctico, do

for cada regla semántica b = f (c1, c2, ..., cn) asociada con la producción del nodo n do

for i:=1 to k do

trazar un arco desde el nodo para ci hasta el nodo para b.

**Orden de Evaluacion**

**Ordenamiento topológico**

Ordenamiento topológico de un grafo acíclico dirigido es todo ordenamiento m1, m2, . . . , mk de los nodos del grafo tal que los arcos vayan desde los nodos que aparecen primero en el ordenamiento a los que aparecen mas tarde.

Todo ordenamiento topológico de un grafo de dependencias da un orden válido en el que se pueden evaluar las reglas semánticas asociadas con los nodos de un árbol de análisis sintáctico.

La **traducción** especificada por una **definicion dirigida por la sintaxis** se puede precicar de la forma sig. :

1.-Construir el arbol de analisis sintactico para la gramatica y la entrada dadas.

2.-Construir explicita o implicitamente el **grafo de dependencias.**

3.-Supuesto que el grafo de dependencias determina un **orden parcial** (esto es cumple las propiedades reflexiva, antisimetrica y transitiva) construir un **orden topologico** compatible con el orden parcial.

4.-Evaluar las reglas semanticas de acuerdo con el orden topologico.

Cadena de entrada

real id1, id2, id3;

orden de evaluacion de las reglas semanticas para la cadena de arriba

a4 = real;

a5 = a4;

añadetipo (id3.entrada,a5);

a7 = a5;

añadetipo (id2.entrada,a7);

a9 = a7;

añadetipo (id1.entrada,a9);

**Arbol sintactico**

Un árbol sintáctico es una version resumida del árbol de análisis sintáctico

Los árboles sintácticos para expresiones aritmeticas tienen

Operadores en nodos internos

Operandos en nodos terminales

|  |  |
| --- | --- |
| E → E1 + T | { E.apn = hazNodo('+', E1.apn, T.apn); } |
| E → E1 - T | { E.apn = hazNodo('-', E1.apn, T.apn); } |
| E → T | { E.apn = T.apn; } |
| T → ( E ) | { T.apn = E.apn; } |
| T → NUM | { T.apn = hazhoja(**num**, **num**.val); } |
| T → **id** | { T.apn = hazhoja(**id** , **id**.entrada); } |

Definicion dirigida por la sintaxis para construir el arbol sintactico de una expresion

Libro del dragón Capitulo 6

**Expresiones de tipo**

El tipo de una construcción de un lenguaje se denotará mediante una “**expresión de tipo**”. De manera informal, una expresión de tipo es, o bien un tipo básico o se forma aplicando un operador llamado constructor de tipos a otras expresiones de tipos.

Una **expresión de tipo** se define recursivamente como:

1.-Un tipo básico es una expresión de tipo. Un tipo básico especial, error\_tipo, señalará un error durante la

comprobación de tipos. Un tipo básico vacío indica “la ausencia de valor”, se usa para comprobar las proposiciones.

2. Como se pueden dar nombre a las expresiones de tipos, el nombre de un tipo es una expresión de tipo (estos nombres se usan por ejemplo cuando se definen registros).

3. Un constructor de tipos aplicado a expresiones de tipos es una expresión de tipo.

4. Las expresiones de tipo pueden contener variables cuyos valores son expresiones de tipos.

PL. Concordancia de tipos

(2)

• Los constructores de tipos incluyen:

a) **Matrices**. Si T es una expresión de tipo, entonces array(I, T) es una expresión de tipo que indica el tipo de una matriz con elementos de tipo T y conjunto de índices I.

b) Productos. Si T 1 y T 2 son expresiones de tipo, entonces su producto cartesiano T 1 ×T 2 es una expresión de tipo.

c) **Registros**. La diferencia entre un registro y un producto es que los campos de un registro tienen nombres. El constructor de tipos record se aplicará a una tupla formada con nombres de campos y tipos de campos.

d) Apuntadores. Si T es una expresión de tipo, entonces pointer(T) es una expresión de tipo que indica el tipo

“apuntador a un objeto de tipo T”.

e) **Funciones**. Se pueden considerar las funciones dentro de los lenguajes de programación como transformaciones de un dominio tipo D a un rango tipo R. La expresión de tipo D→R indicará el tipo de dicha función

**Sistema de tipos**

Un **sistema de tipos** es una serie de reglas para asignar expresiones de tipos a las distintas partes de un programa. Un comprobador de tipos implanta un sistema de tipos.

**Comprobación estática y dinámica de tipos**

Los sistemas de tipos se pueden especificar mediante una d.d.s.

• Se dice que la comprobación realizada por un compilador es **estática**, mientras que la comprobación hecha al ejecutar el programa objeto se denomina **dinámica**. En principio, cualquier verificación se puede realizar dinámicamente, si el código objeto carga el tipo de un elemento junto con el valor de dicho elemento.

• Un **sistema de tipos seguro** elimina la necesidad de comprobar dinámicamente errores de tipos ya que permite determinar estáticamente que dichos errores no pueden ocurrir cuando se está ejecutando el programa objeto.

• Se dice que un **lenguaje** es **fuertemente tipado** si su compilador puede garantizar que los programas que acepte se ejecutarán sin errores de tipo. En la práctica, algunas comprobaciones sólo se pueden hacer dinámicamente.

**Especificacion de un comprobador de tipos sencillo**

**Comprobador de tipos**

El comprobador de tipos es un esquema de traducción que sintetiza el tipo de cada expresión a partir de los tipos de sus subexpresiones.

**Un lenguaje simple**

Gramática para lenguaje fuente

P→D; E

D→D; D | id : T

T→ **char** | **integer** | **array** [ **núm** ] of T | ^T

E→ literal | núm | id | E mod E | E [ E ] | E ^

|  |  |
| --- | --- |
| P→D; E |  |
| D→D; D |  |
| D→ **id** : T | { añadetipo(**id**.entrada, T.tipo) } |
| T→ **char** | { T.tipo := char } |
| T→ **integer** | { T.tipo := integer } |
| T→ ^T1 | { T.tipo := pointer(T1.tipo) } |
| T→ **array** [ **núm** ] **of** T1 | { T.tipo := array(1..**núm**.val, T1.tipo ) } |

811

P → { desplazamiento := 0 } D

D → D ; D

D → id : T { introduce( id.name, T.tipo , desplazamiento );

desplazamiento := desplazamiento + T.ancho }

T → integer { T. tipo := integer; T. ancho := 4 }

T → real { T. tipo := real; T. ancho := 8 }

T → array [ num ] of T1 { T. tipo := array( num.val, T1. tipo );

T. ancho := num.val \* T1. ancho }

T → ^ T1 { T. tipo := pointer( T1.type ); T. ancho := 4 }

Esquema de traduccion

**Comprobación de tipos en las expresiones**

|  |  |
| --- | --- |
| E → **literal** | { E. tipo := char } |
| E → **num** | { E. tipo := integer } |
| E → **id** | { E.tipo:= busca(**id**.entrada) } |
| E → E1 + E2 | { E. tipo := **if** E1 .tipo = integer **and** E2 ..tipo = integer **then** integer  **else** error\_tipo } |
| E → E1 [ E2 ] | { E. tipo := **if** E2 .tipo = integer **and** E1 .tipo = array (s, t) **then** t  **else** error\_tipo } |
| E→ E ^ | { E. tipo := **if** E1 .tipo = pointer( t ) **then** t  **else** error\_tipo } |
| E → id | E.lugar:=id.lugar |

**Comprobación de tipos en las proposiciones**

|  |  |
| --- | --- |
| S → id := E | { S. tipo := **if** id.tipo = E.tipo **then** vacio  **else** error\_tipo } |
| S→ **if** E **then** S1 | { S. tipo :=  **if** E.tipo = boolean **then** S1. tipo  **else** error\_tipo } |
| S→ **while** E **do** S1 | { S. tipo := **if** E.tipo = boolean **then** S1. tipo  **else** error\_tipo } |
| S→S1; S2 | { S. tipo := **if** S1 .tipo = vacio **and** S2 .tipo = vacio **then** vacio  **else** error\_tipo } |

**Comprobación de tipos de funciones**

La aplicación de una función a un argumento puede ser representada por la producción

E → E (E)

en la que una expresion es la aplicación de una expresion a otra. Se pueden aumentar las reglas para asociar expresiones de tipos con el no terminal T con la siguiente producción y acción para permitir tipos de función en las declaraciones.

T → T1 '→' T2 {T tipo:= T1.tipo→ T2.tipo }

Las comillas alrededor de la flecha que se usa como constructor de función lo distinguen de la flecha que se usa como metasimbolo en una producción.

La regla para comprobar el tipo de aplicación de una función es

E → E1 ( E2 ) { E. tipo := **if** E2 .tipo = s **and** E1 .tipo = s → t **then** t **else** error\_tipo }

Esta regla indica que en una expresión formada por la aplicación de E1 a E2 , el tipo de E1 debe ser una función s → t del tipo s de E2  a algun tipo de rango t; el tipo de E1 ( E2 ) es t.

dos tipos son iguales si sus estructuras son iguales

**Equivalencia de expresiones de tipo**

**Equivalencia estructural de expresiones de tipo**

**Equivalencia estructural**: cuando dos expresiones de tipos son, o bien el mismo tipo básico, o están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes. Un algoritmo que comprueba esto es:

boolean equivest (s, t){

if (s y t son el mismo tipo básico)

return true

else if (s == array(s 1 , s 2 ) && t== array(t 1 , t 2 ) )

return equivest(s 1 , t 1 ) && equivest(s 2 , t 2 )

else if (s == s 1 ×s 2 and t == t 1 ×t 2 )

return equivest(s 1 , t 1 ) && equivest(s 2 , t 2 )

else if (s == pointer(s 1 ) && t == pointer(t 1 ) )

return equivest(s 1 , t 1 )

else if (s == s 1 →s 2 && t == t 1 →t 2 )

return equivest(s 1 , t 1 ) && equivest(s 2 , t 2 )

else

return false

}

**Nombres para expresiones de tipos**:

Se permite dar nombres a las expresiones de tipo y que estos nombres aparezcan en expresiones de tipos donde previamente sólo existían tipos básicos

Equivalencia de nombre: dos expresiones de tipo tienen equivalencia de nombre, si y sólo si, son idénticas

Equivalencia estructural: dos expresiones de tipos son estructuralmente equivalentes si representan dos

expresiones de tipos estructuralmente equivalentes cuando todos los nombres han sido sustituidos

**Conversiones de tipos.**

En expresiones, la transformación más común es la de convertir el entero en un numero real y después realizar una operación real con el par de operandos reales obtenidos.

Coerciones Se dice que la conversión de un tipo a otro es implícita si el compilador la va a realizar automáticamente

**SOBRECARGA DE** **FUNCIONES Y OPERADORES**

Un símbolo sobrecargado es el que tiene distintos significados dependiendo de su contexto. En matemáticas, el operador de suma + está sobrecargado, porque + en A + B tiene distintos significados si A y B son enteros, reales, números complejos o matrices.

La sobrecarga se resuelve cuando se determina un significado único para un caso de símbolo sobrecargado

La resolución de sobrecarga se denomina *identificación de operadores*

Una **función polimórfica** es aquella que puede ejecutarse con argumentos de tipos distintos

**Variables de tipo**

Las variables que representan expresiones de tipos permiten considerar tipos desconocidos

*Inferencia de tipos*: problema de determinar el tipo de una construcción del lenguaje a partir del modo en que se usa.

La función longitud sigue los siguientes enlaces dentro de la lista hasta que se encuentra un enlace con valor nil.

type enlace = nodo;

nodo = record info : integer;

siguiente : enlace

end;

function longitud ( aplista : enlace ) : integer;

var lon : integer;

begin

lon : = 0;

while aplista <> nil do begin

lon : = lon + l;

aplista := aplista. siguiente

end;

longitud : = lon

end;

El procedimiento mlista aplica el parámetro p a todos los nodos de una lista enlazada.

type enlace = nodo;

procedure mlista ( aplista : enlace; procedure p ) ;

begin

while aplista <> nil do begin

p (aplista);

aplista : = aplista. siguiente

end

end;

function desref(p);

begin

return p↑

end;

• El tipo de la función es ∀α. pointer(α) →α

• Una expresión de tipo que contenga un símbolo ∀ se considerará un “tipo polimórfico”

Para la función que retorna la longitud de una lista:

∀ α . List(α ) → integer

La equivalencia de tipos se determina haciendo unificación de tipos.

**Un lenguaje con funciones polimórficas**

P → D ; E

D → D ; D | id : Q

Q → ∀ variable\_de\_tipo.Q | T

T → T ‘→’ T

| T × T

| constructor\_unario ( T )

| tipo\_basico

| variable\_de\_tipo

| ( T )

E → E ( E ) | E , E | id

Ejemplo

desref: ∀α.pointer(α) →α

q: pointer(pointer(integer)

desref(desref(q))

**Sustituciones, casos y unificación**

La siguiente función recursiva sust(t) precisa la noción de aplicar una sustitución S para sustituir todas las variables de tipo en una expresión t. Como de costumbre, se considera el constructor de tipos de función como el constructor “típico”.

function sust (t:expresión\_tipo):expresión\_tipo;

begin

if t es un tipo básico then return t

else if t es una variable then return S(t)

else if t es t1→t2 then return sust(t1) → sust(t2)

end

Para simplificar, se escribe S(t) para la expresión de tipo que se obtiene cuando se aplica sust a t; el resultado S (t) se denomina caso de t.

En lo sucesivo, se escribirá s<t para indicar que s es un caso de t:

pointer(integer) < pointer(α)

**Comprobación de funciones polimórficas**

Las reglas para comprobar expresiones generadas por la gramatica para unlenguaje con funciones polimórficas se escribiran se escribiran mediante las siguientes operaciones en una representacion con grafos de los tipos.

1. nuevas (t) sustituye las variables acotadas en la expresión de tipo t por variables nuevas y devuelve un apuntador a un nodo que representa la expresión de tipo obtenida. En el proceso se eliminan todos los simbolos ∀ en t.

2. unifica (m, n) unifica las expresiones de tipos representadas por los nodos apuntados por m y n. Como efecto secundario conserva la sustitución que convierte las expresiónes en equivalentes. Si falla el proceso de unificación de las expresiones, falla todo el proceso de comprobación de tipos.

|  |  |
| --- | --- |
| E → E1 ( E2 ) | { p:= hazhoja (vartiponueva);  unifica (E1.tipo, haznodo (“→”, E2.tipo, p));  E.tipo:=p; } |
| E → El , E2 | { Etipo:= haznodo ('X', El.tipo, E2.tipo)} |
| E → id | { E.tipo:= nuevas(**id**.tipo) } |

Esquema de traducción para la comprobación de funciones polimórficas

**UNALGORITMO PARA LA UNIFICACION**

De manera informal, la unificación es el problema de determinar si dos expresiones e y f pueden convertirse en identicas sustituyendo expresiones por las variables que aparezcan en e y f. Comprobar la igualdad de expresiones es un caso especial de unificación si e y f tienen constantes pero no variables, entonces e y fse unifican si, y sólo si, son identicas. El algoritmo de unificación de esta sección puede aplicarse a grafos con ciclos, así que se puede utilizar para comprobar si hay equivalencia estructural en tipos circulares.

La unificación fue definida en la última sección en términos de una función S, llamada sustitución, que transforma variables en expresiones. Se escribe S(e) para la expresión que se obtiene al sustituir cada variable α en e por S(α)

El algoritmo de esta sección determina una sustitución que es el unificador mas general un par de expresiones.

Ejemplo 6.13. Para tener una perspectiva de los unificadores más generales, considérense las dos expresiones de tipos

((α1 → α2 ) x list (α3)) → list(α2)

((α3 → α4) x list (α3 )) → α5

Dos unificadores, S y S", para estas expresiones son

|  |  |  |
| --- | --- | --- |
| x | S(x) | S'(x) |
| α1 | α3 | α1 |
| α2 | α2 | α1 |
| α3 | α3 | α1 |
| α4 | α2 | α1 |
| α5 | list (α2) | list (α1) |

Dos expresiones de tipo t1 y t2 unifican, si existe una sustitución S tal que S(t1) = S(t2).

En la practica se busca encontrar el unificador mas general, es decir, la sustitución que establece el menor numero de restricciones posible sobre las variables de las expresiones.

El unificador mas general de t1 y t2 es una sustitución que cumple:

S(t1) = S(t2)

∀S′ tal que S′(t1) = S′(t2) se tiene que S′ es un cado de S.

Algoritmo 6.1. Unificación de un par de nodos en

Entrada : Un grafo y un par de nodos m y n que deben unificarse

Salida:

**function** unifica (m, n, nodo): boolean;

**begin**

s:= encuentra(m);

t:= encuentra(n);

if s = t **then**

**return true**

**else** **if** sy t son nodos que representan al mismo tipo básico **then**

**return true**

**else** **if** s es un nodo interior con hijos s1 y s2 **and** t un nodo interior con hijos t1 y t2 **then begin**

union(s,t);

**return** unifica (s1, t1) **and** unifica (s2, t2)

**end**

**else** **if** so t representan a una variable **then begin**

union(s,t);

**return true**

**end**

**else** **return false**

**end**

1. encuentra (n) devuelve el nodo representante de la clase de equivalencia que en ese momento contenga el nodo n.

2. unión(m, n) fusiona las clases de equivalencia que contengan los nodos m y n. Si uno de los representantes de las clases de equivalencia de m y n es un nodo no variable, unión convierte al nodo no variable en representante de la clase de la clase de equivalencia fusionada de lo contrario, unión convierte a uno o al otro de los representantes originales en el nuevo representante. Esta asimetría en la especificación de unión es importante porque una variable no se puede utilizar representante de una clase de equivalencia para una expresion que contenga un constructor de tipos o un tipo básico . De lo contrario dos expresiones no equivalentes se podrian unificar por medio de esa variable.

Libro del dragón Capitulo 7

**Definición de un** Procedimiento

Es un conjunto de instrucciones a las que se asocia un nombre

Es un bloque de codigo al que se asocia un nombre

Es una declaración que asocia un identificador con una proposicion.

**Arboles de activacion (Arboles de llamadas)**

1.-El control fluye secuencialmente, la ejecución de un programa consta de una secuencia de pasos, en cada paso el control está en un punto específico del programa.

2.-Cada ejecución de un procedimiento inicia al principio del cuerpo del procedimiento, y termina cuando se retorna el control al punto inmediatamente siguiente a aquel donde el procedimiento fue llamado

La ejecución del cuerpo de un procedimiento es una activación del procedimiento. El tiempo de vida de una activación de un procedimiento, es la secuencia de pasos que transcurre entre el primero y el último paso en la ejecución del cuerpo del procedimiento.

Asumiendo lo anterior, si a y b son activaciones de procedimientos, entonces sus tiempos de vida: No se sobreponen o Son anidados.

**Recursividad**

Un procedimiento es recursivo si una activación del procedimiento puede iniciar antes que finalice una activación anterior del mismo.

Un procedimiento es recursivo si se llama a si mismo en su definicion

**Arbol de Activación ( Arboles de llamadas )**

Arbol que representa la forma en que el control entra y deja las activaciones

Cada nodo representa la activación de un procedimiento (una unica activación)

La raiz representa la activación del procedimiento principal (main)

El nodo a es padre del nodo b, si y solo si el control fluye de una activación de a a b.

El nodo a esta a la izquierda del nodo b, si y solo si el tiempo de vida de a ocurre antes que el tiempo de vida de b

**Pila de Control**

El control de flujo de un programa corresponde a un recorrido en profundidad del árbol de activación, visita el nodo antes que sus hijos recursivamente visita los hijos de izquierda a derecha

Se usa una Pila de Control para llevar el rastro delos procedimientos activos, el nodo que corresponde a una activación entra en la pila cuando la activación inicia, y sale de la misma cuando la activación termina.

**Ámbito de una declaración**

Una declaración es una construcción sintáctica que asocia un nombre con información. Pueden ser explicitas o implícitas. El mismo nombre puede ser declarado en diferentes partes del programa. Las Reglas de Alcance de un programa determinan cual declaración de un nombre aplica cuando el nombre es usado en el programa.

El ambito de una declaración es la parte del programa en la cual la declaración aplica. Dentro de un procedimiento los nombres pueden ser Locales o No-Locales.

**Enlace de Nombres**

**Ambiente**: una función que mapea un nombre con una posición de almacenamiento

**Estado**: una función que mapea una posición de almacenamiento con el valor que contiene.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  |  | Ambiente |  | Estado |  |
|  |  |  | → |  | → |  |
|  |  | Nombre |  | memoria |  | Valor |

Una transformacion de dos etapas de nombres a valores

A esto se refiere también la distinción del significado de un identificador al lado izquierdo o derecho de una asignación:

R-value: el lado derecho significa el valor (Estado)

L-value: el lado izquierdo la posición de almacenamiento (Ambiente)

**Ambiente** transforma un nombre en un valor de lado izquierdo

**Estado** transforma el valor de lado izquierdo en un valor de lado derecho

Cuando el ambiente enlaza una posición de almacenamiento s con un nombre x, se dice que x esta enlazado a s.

**Enlace** es la contrapartidadinámica de una declaración

Estático vs. Dinámico:

|  |  |
| --- | --- |
| estática | dinámica |
| definición de un procedimiento | activaciones de un procedimiento |
| declaración de un nombre | enlaces del nombre |
| ámbito de una declaración | duración enlace |

**7.2 Organizacion de la memoria**

**Registros de activacion ( marco )**

Es la informacion necesaria para una sola ejecucion de un procedimiento.

Informacion importante que se almacena en un marco

-Parametros reales (actuales)

-Direccion de retorno ( relacionada con el valor del contador de programa)

-Variables locales

Para lenguaje C normalmente se introduce el registro de activacion (marco ) de un procedimiento en la pila de ejecucion (pila de llamadas) cuando se llama al procedimiento.y se saca de la pila cuando el control regresa al autor de la llamada.

7.3 **Estrategias para la asignacion de memoria**

La asignación por medio de una Pila trata la memoria en ejecucion como una pila.

**La asignación por medio de una Pila**

Dado que los registros de activacion se introducen y se sacan de la pila cuando las llamadas comienzan y terminan respectivamente se deduce que las variables locales en una activacion se “crean” y los valores de dichas variables se “destruyen” cuando finaliza la activacion.

Secuencias de Llamado y Retorno

La **secuencia de llamada** a procedimientos se genera en el código destino. Debe asignar el espacio de memoria necesario para su activación e ingresar la información en sus campos.

La **secuencia de retorno** restaura el estado de la máquina de tal forma que el procedimiento llamador pueda continuar su ejecución

|  |
| --- |
| Parámetros Actuales y valor de Retorno |
| Estado guardado y enlaces |
| Temporales y datos Locales |
| Parámetros Actuales y valor de Retorno |
| Estado guardado y enlaces |
| Temporales y datos Locales |
|  |
| registro de activacion general |

La secuencia de Llamada

El procedimiento Llamador:

1.-Evalúa los parámetros actuales

2.-Guarda la dirección de retorno y el valor del tope de la pila

El procedimiento Llamado:

Incrementa el apuntador del tope de la pila

3.-Guarda el estado de la máquina

4.-Inicializa los datos locales e inicia la ejecución

Una posible secuencia de retorno seria:

El procedimiento Llamado:

1.-Ubica el valor de retorno

2.-Restaura el tope de la pila y el estado de la máquina

El procedimiento Llamado:

3.-Copia el valor de retorno a su propio registro de activación

**7.4 Acceso a nombres no locales**

**bloques**

Un bloque es una proposicion que contiene sus propias declaraciones de datos locales. Una caracteristica de los bloques es su estructura de anidamiento.

Regla de anidamiento mas cercano.

1.-El ambito de una declaración en un bloque B incluye B.

2.-Si un nombre x no esta declarado en un bloque B, entonces un caso de x en B esta en el ambito de una declaración de x en un bloque abarcador tal que

i) tiene una declaración de x, y

ii) esta anidado mas cerca alrededor de B que cualquier otro bloque con una declaración de x.

**7.5 Paso de parametros**

Cuando un procedimiento llama a otro, se comunican habitualmente por medio de nombres no locales y de parametros delprocedimiento llamado.

**Llamada por valor**

1.-Un parametro formal se considera como un nombre local de modo que las direcciones de memoria para los parametros formales se encuentran en el registro de activacion delprocedimiento llamado.

2.-El procedimiento autor de la llamada evalúa los parámetros actuales y coloca sus valores de lado derecho en las direcciones de memoria de los parametros formales.

**Llamada por referencia**

Cuando los parámetros se pasan por referenciaelautor de la llamada pasa al procedimiento llamado un apuntador a la direccion de memoria de cada parámetro actual.

1.-Si un parámetro actual es un nombre o una expresion que tenga un valor de lado izquierdo, entonces se pasa ese mismo valor de lado izquierdo.

2.- Sin embargo si el parámetro actual es una expresion como a+b o 2 que no tiene ningun valor de lado izquierdo entonces la expresion se evalúa en una nueva posicion y se pasa la direccion de dicha posicion.

**7.6 Tabla de simbolos**

**Entradas de la tabla de simbolos**

Cada entrada en la tabla de simbolos corresponde a la declaracion de un nombre. Si la declaracion es de una variable se puede guardar en la entrada correspondiente su nombre y su tipo.

Las palabras clave se introducen al inicio antes de que comience el analisis lexico.

**Estructura de datos de tipo lista para las tablas de simbolos**

funcion de **busqueda** de un nombre en la lista usada para implementar la tabla de simbolos. Si lo encuentra regresa la entrada correspondiente a ese nombre y si no lo encuentra regresa nulo.

funcion para **agregar** a la tabla de simbolos una entrada para un nombre**.** Dichafuncion puede agregar dicha entrada al inicio de la lista.

**Tablas de dispersion**

1.Tabla de dispersion formada por un arreglo fijo de m apuntadores a entradas de la tabla.

2. Entradas de la tabla organizadas en m listas enlazadas independientes llamadas cubetas

**funciones de dispersion**

recibe como parametro una cadena y devuelve un entero entre 0 y m-1

Para calcular la funcion de dispersion se hace lo siguiente:

1. Determinar un entero positivo a partir de los carac de la cadena

2. Para indexar el arreglo se usa el residuo que resulta de dividir el entero positivo entre m.

7.7 **Instrumentos de los lenguajes para la asignacion dinamica de la memoria**

**Basura**

La memoria que un programa asigna, pero a la que no puede referirse, se denomina **basura**

LISP realiza recolecion de basura . Pascal y C no realizan recolecion de basura dejando que el programador desasigne explicitamente la memoria que ya no sirve.

**Referencias suspendidas**

Se produce cuando se hace referenciaamemoria desasignada.

Las referencias suspendidas y la basura son conceptos relacionados; las referencias suspendidas ocurren si se produce la desasignacion antes de la ultima referencia mientras que existe basura si se produce la ultima referencia antes de la desasignacion.

**7.8 Tecnicas de asignacion dinamica de memoria**

Las tecicas necesarias para implantar la asignacion dinamica de memoria dependen de como se desasigne la memoria.Si la desasignacion es implicita entonces el paquete de apoyo para la ejecucion es el encargado de determinar cuando un bloque de memoria ya no es necesario. Si la desasignacion se realiza explicitamente por el programador, el compilador tiene menos trabajo

Asignacion explicita de bloques de tamaño fijo

Enlazando los bloques en una lista se puede hacer rapidamente la asignacion y la desasignacion con poca o nula perdida de memoria.

La asignacion consiste en sacar un bloque de la lista y la desasignacion en devolver un bloque a la lista

Asignacion explicita de bloques de tamaño variable

Cuando se asignan y desasignan bloques la memoria se puede fragmentar

Un metodo para asignar bloques de tamaño variable es el metodo de **primer ajuste.** Cuandoseasigna un bloque se busca el primer bloque libre con el tamaño suficiente. Entonces el bloque se divide en dos fragmentos, uno para el proceso y otro para la memoria desocupada.

Cuando se desasigna un bloque se checa que sea contiguo a un bloque libre. Si es posible el bloque desasignado se combina con el bloque libre contiguo a el para crear un bloque libre mas grande. Combinar bloques libres adyacentes en un bloque libre mayor evita que se produzcan mas fragmentaciones

Desasignacion implicita

reconocer fronteras del bloque

reconocer bloque esta en uso

Un bloque esta en uso si es posible que el programa haga referencia a la informacion contenida en el bloque.

1. Cuenta de referencias. Se lleva la cuenta del numero de bloques que apuntan directamente al presente bloque. Cuando la cuenta baja a 0 se puede desasignar el bloque porque no se le puede hacer referencia. Es decir el bloque se ha convertido en basura que puede recolectarse. Mantener la cuenta de las referencias puede suponer mucho tiempo. La asignacion p:=q cambia la cuenta de referencias de los bloques apuntados por p y q. La cuenta para el bloque apuntado por p disminuye en uno, mientras que la cuenta para el bloque apuntado por q aumenta en uno. Las cuenta de referencias se utilizan mejor cuando los apuntadores entre bloques nunca aparecen en ciclos.

2. Tecnicas de marca

Libro del dragón Capitulo 8

Los códigos intermedios pueden ser:

–Árbol sintáctico

-Notación Posfija

-Código de tres direcciones

**Árbol sintáctico o Arbol de sintaxis abstracta**

Cada hoja representa un operando y los nodos internos representan los operadores.

**Notación Posfija**

– notación matemática libre de paréntesis

– los operadores aparecen después de los operandos

**Código de tres direcciones**

Es una especie de código máquina de la forma general:

x:=y op z

**Tipos de proposiciones de tres direcciones**

1. Asignación de operadores binarios:

• x:=y op z

2. Asignación de operadores unarios:

• x:=op y

3. Proposiciones de copia:

• x:=y

4. Salto incondicional:

• goto E

5. Saltos condicionales:

• if x oprel y goto E

6. Llamadas a procedimientos:

• param x

• call p, n

• return y

7. Asignaciones con índices:

• x:=y[i]

• x[i]:=y

8. Asignaciones de direcciones y apuntadores:

• x:=&y

• x:=\*y

• \*x:=y

|  |  |
| --- | --- |
|  | 8.6 |
| S → id := E | S.codigo := E.codigo || emite( id. lugar ‘:=‘ E. lugar ) |
| E → E1 + E2 | { E.lugar:=tempnuevo();  E.codigo := E1 .codigo || E2 .codigo ||  emite(E.lugar '=' E1 .lugar '+' E2 .lugar ); } |
| E → E1 \* E2 | { E.lugar:=tempnuevo();  E.codigo := E1 .codigo || E2 .codigo ||  emite(E.lugar '=' E1 .lugar '\*' E2 .lugar ); } |
| E → - E1 | { E.lugar:=tempnuevo();  E.codigo := E1 .codigo || emite(E.lugar ':=' 'menosu' E1.lugar) } |
| E → ( E1 ) | { E.lugar:=E1 .lugar  E.codigo := E1 .codigo } |
| E → id | E.lugar:=id.lugar |

**Declaraciones**

Las declaraciones de variables no generan código.

• La declaración de una variable debe crear una entrada en la tabla de símbolos.

• Esta información incluye el identificador de la variable, el tipo de dato (que permite calcular su tamaño) y su posición de memoria (desplazamiento respecto a una posición conocida)

811

P → { desplazamiento := 0 } D

D → D **;** D

D → **id** **:** T { introduce( id.name, T.tipo , desplazamiento );

desplazamiento := desplazamiento + T.ancho }

T → **integer** { T. tipo := integer; T. ancho := 4 }

T → **real** { T. tipo := real; T. ancho := 8 }

T → **array** **[ num ]** **of** T1 { T. tipo := array( num.val, T1. tipo );

T. ancho := num.val \* T1. ancho }

T → **^** T1 { T. tipo := pointer( T1.type ); T. ancho := 4 }

813 procedimientos anidados

Cuando se un procedimiento anidado declaraciones procedimiento abarcador

P -> D

D -> D ; D | id : T | proc id ; D ; S

S→ id:= E { emite(id.lugar ':=' E.lugar) }

**Proposiciones de asignación 815**

|  |  |
| --- | --- |
| S → id := E | { emite(id.lugar ':=' E.lugar) } |
| E → E1 + E2 | { E. lugar = tempnvo();  emite(E.lugar '=' E1 .lugar '+' E2 .lugar ); } |
| E → E1 \* E2 | { E. lugar = tempnvo();  emite(E.lugar '=' E1 .lugar '\*' E2 .lugar );} |
| E → - E1 | { E.lugar:=tempnuevo(); emite(E.lugar ':=' 'menosu' E1.lugar) } |
| E → ( E1 ) | { E.lugar:=E1 .lugar } |
| E → id |  |

**Acceso a elementos de matrices**

matriz unidimensional

i x a + base

matriz bidimensional

( i1n2 + i2 ) x a + base

matriz multidimensional

. (( ... ( i1n2 + i2 )n3 + i3 ...)nk +ik) x a + base

recurrencia

e1 = i1

em = em-1n2 + i2

|  |  |
| --- | --- |
|  | 8.2 |
|  |  |
| S -> L := E | { if L.offset = null then /\* L is a simple id \*/  emite(L.lugar ‘:=‘ E.lugar );  else  emite(L.lugar ’[‘ L.offset ‘]’ ‘:=‘ E.lugar ) } |
| E → E1 + E2 | { E. lugar = tempnvo();  emite(E.lugar ':=' E1 . lugar '+' E2 .lugar ); } |
| E → ( E1 ) | E.lugar:=E1 .lugar |
| E -> L | { if L.offset = null then /\* L is a simple id \*/  E. lugar := L. lugar  else begin  E. lugar := tempnvo();  emite( E. lugar ‘:=‘ L. lugar ‘[‘ L.offset ‘]’ )  end } |
| L -> Elist ] | { L. lugar := tempnvo();  L.offset := tempnvo();  emite(L. lugar ‘:=‘ c(Elist.matriz ));  emite(L.offset ‘:=‘ Elist. lugar ‘\*’ width(Elist.matriz )) } |
| L -> id | { L. lugar := id.lugar ; L.offset = null } |
| Elist -> Elist1, E | { t := tempnvo(); m := Elist1.ndim + 1;  emite(t ‘:=‘ Elist1. lugar ‘\*’ limit( Elist1.matriz , m ));  emite(t ‘:=‘ t ‘+’ E. lugar );  Elist. matriz := Elist1.matriz ;  Elist. lugar := t; Elist.ndim := m } |
| Elist -> id [ E | { Elist.matriz := id.lugar ;  Elist. lugar := E.lugar ; Elist.ndim := 1 } |

x:=A[y, z] .

La asignación se traduce a la siguiente secuencia de proposiciones de tres direcciones:

t1 : = y \* 20

t1 : = t1+z

t2 := c

t3 := 4 \* t1

t4 := t2 [ t3]

x := t4

|  |  |
| --- | --- |
|  | 8.2 |
|  |  |
| S -> L := E | { if L.offset = null then /\* L is a simple id \*/  emite(L.lugar ‘:=‘ E.lugar );  else  emite(L.lugar ’[‘ L.offset ‘]’ ‘:=‘ E.lugar ) } |
| E → E1 + E2 | { E. lugar = tempnvo();  emite(E.lugar ':=' E1 . lugar '+' E2 .lugar ); } |
| E → ( E1 ) | E.lugar:=E1 .lugar |
| E -> L | { if L.offset = null then /\* L is a simple id \*/  E. lugar := L. lugar  else begin  E. lugar := tempnvo();  emite( E. lugar ‘:=‘ L. lugar ‘[‘ L.offset ‘]’ )  end } |
| L -> id [ E ] | { L. lugar := tempnvo();  L.offset := tempnvo();  emite(L. lugar ‘:=‘ c(Elist.matriz ));  emite(L.offset ‘:=‘ Elist. lugar ‘\*’ width(Elist.matriz )) } |
| L -> L [ E ] | { t := tempnvo(); m := L1.ndim + 1;  emite(t ‘:=‘ L1. lugar ‘\*’ limit( L1.matriz , m ));  emite(t ‘:=‘ t ‘+’ E. lugar );  L. matriz := L1.matriz ;  L. lugar := t; L.ndim := m } |
| L -> id | { L. lugar := id.lugar ; L.offset = null } |
| Elist -> Elist1, E | { t := tempnvo(); m := Elist1.ndim + 1;  emite(t ‘:=‘ Elist1. lugar ‘\*’ limit( Elist1.matriz , m ));  emite(t ‘:=‘ t ‘+’ E. lugar );  Elist. matriz := Elist1.matriz ;  Elist. lugar := t; Elist.ndim := m } |
| Elist -> id [ E | { Elist.matriz := id.lugar ;  Elist. lugar := E.lugar ; Elist.ndim := 1 } |

Esquema de traducciónpara producir código de tres direcciones para las asignaciones

**Expresiones booleanas**

Representation Numerica

820

|  |  |
| --- | --- |
|  |  |
| E → E1 **or** E2 | { E.lugar:=tempnuevo();  emite(E.lugar '=' E1 .lugar 'or' E2 .lugar ); } |
| E → E1 **and** E2 | { E.lugar:=tempnuevo();  emite(E.lugar '=' E1 .lugar 'or' E2 .lugar ); } |
| E → **not** E1 | { E.lugar:=tempnuevo();  emite(E.lugar ':=' 'not' E1.lugar) } |
| E → ( E1 ) | { E.lugar:=E1 .lugar } |
| E → **id**1 **oprel** **id**2 | { E.lugar:=tempnuevo();  emite (‘**if**’ **id1**.lugar **oprel**.op **id2**.lugar ‘goto’ sigte prop + 3);  emite (E.lugar ‘:=‘ ‘0’);  emite (‘**goto**’ sigte prop + 2);  emite (E.lugar ‘:=‘ ‘1’);} |
| E →**true** | { E.lugar:=tempnuevo();  emite(E.lugar ':=' '1' ) } |
| E →**false** | { E.lugar:=tempnuevo();  emite(E.lugar ':=' '0' ) } |

821

|  |  |  |  |
| --- | --- | --- | --- |
| 100 | if a < b goto 103 | 107 | t2 := 1 |
| 101 | t1: = 0 | 108 | if e < f goto 111 |
| 102 | goto 104 | 109 | t3: = 0 |
| 103 | t1 := 1 | 110 | goto 112 |
| 104 | if c < d goto 107 | 111 | t3 := 1 |
| 105 | t2 : = 0 | 112 | t4 : = t2 and t3 |
| 106 | goto 108 | 113 | t5 : = t1 or t4 |
| Traduccion de a < b || c < d && e < f | | | |

**Codigo en corto circuito**

S→ if { I.falso:=etiqnueva(); } E then { emite('if' E.lugar '=' '0' 'goto' I.falso); } S1 I

823

|  |  |
| --- | --- |
|  |  |
| S→ **if** E **then** S1 | E.verdadera:=etiqnueva();  E.falsa:=S.siguiente;  S1.siguiente=S.siguiente;  S.codigo := E.codigo ||  emite(E.verdadera ':' ) || S1 .codigo |
| S→ **if** E **then** S1 **else** S2 | E.verdadera:=etiqnueva();  E.falsa:=etiqnueva();  S1.siguiente=S.siguiente;  S2.siguiente=S.siguiente;  S.codigo := E.codigo ||  emite(E.verdadera ':' ) || S1 .codigo ||  emite( 'goto ' S.siguiente) ||  emite(E.falsa':' )|| S2.codigo |
| S→ **while** E **do** S1 | S.comienzo:=etiqnueva();  E.verdadera:=etiqnueva();  E.falsa:=S.siguiente;  S1.siguiente:=S.comienzo;  S.codigo:= emite(S.comienzo':') || E.codigo ||  emite(E.verdadera ':' ) || S1 .codigo ||  emite( 'goto ' S.comienzo) |

824

|  |  |
| --- | --- |
|  |  |
| E → E1 **or** E2 | E1.verdadera:=E.verdadera;  E1.falsa:=etiqnueva();  E2.verdadera:=E.verdadera;  E2.falsa:=E.falsa;  E.codigo := E1 .codigo || emite( 'goto ' E.falsa) || E2 .codigo |
| E → E1 **and** E2 | E1.verdadera:=etiqnueva();  E1.falsa:=E.falsa;  E2.verdadera:=E.verdadera;  E2.falsa:=E.falsa;  E.codigo := E1 .codigo || emite( 'goto ' E.verdadera ) || E2 .codigo |
| E → **not** E1 | E1.verdadera:=E.falsa ;  E2.falsa:=E.verdadera;  E.codigo := E1 .codigo |
| E → ( E1 ) | E1.verdadera:=E.verdadera;  E2.falsa:=E.falsa;  E.codigo := E1 .codigo |
| E → **id**1 **oprel** **id**2 | E.codigo :=emite (‘**if**’ **id1**.lugar **oprel**.op **id2**.lugar ‘goto’ E.verdadera) ||  emite (‘**goto**’ E.falsa) |
| E →**true** | E.codigo := emite( 'goto ' E.verdadera ) |
| E →**false** | E.codigo := emite( 'goto ' E.falsa) |

Ejemplo

a < b || c < d && e < f

|  |  |
| --- | --- |
|  | if a < b goto Lverdadera |
|  | goto L1 |
| L1: | if c < d goto L2 |
|  | goto Lfalsa |
| L2: | if e < f goto Lverdadera |
|  | goto Lfalsa |

Llamadas a procedimientos

S→ call **id** ( listaE )

listaE→ listaE , E

listaE→ E

secuencia de llamadas

S→ CALL VAR ( arglist ) {

emite(call ); //exit(0);

}

Libro del dragón Capitulo 9

**SELECCIÓN DE INSTRUCCIONES**

La naturaleza del conjunto de instrucciones de la máquina objeto determina la dificultad de la selección de instrucciones. Es importante que el conjunto de instrucciones sea uniforme y completo. Si la máquina objeto no apoya cada tipo de datos de una manera uniforme, entonces cada excepción a la regla general exige un tratamiento especial.

Las velocidades de las instrucciones y las expresiones particulares de la máquina son otros factores importantes. Para cada tipo de proposición de tres direcciones, se puede diseñar un esqueleto de código que perfila el código objeto que ha de generarse para esa construcción.

La calidad del código generado viene determinada por su velocidad y tamaño. Las velocidades de las instrucciones son necesarias para diseñar nuevas secuencias de código pero es difícil obtener información exacta de los tiempos de ejecución. Decidir cual es la mejor secuencia de código de máquina para una determinada construcción de tres direcciones también puede exigir conocer el contexto en el que aparece esa construcción. Para cada tipo de proposición de tres direcciones, se puede diseñar un esqueleto de código.

Ejemplo: código de tres direcciones de:

x := y + z

MOV y, R0 /\* cargar y en el registro R0 \*/

ADD z, R0 /\* suma z a R0 \*/

MOV R0, x /\* almacenar R0 en x \*/

**ASIGNACIÓN DE REGISTROS**

Las instrucciones que implican operandos en registros son más cortas y rápidas que las de operandos en memoria. El uso de registros se divide en 2 subproblemas:

1. Durante la asignación de los registros, se selecciona el conjunto de variables que residirá en los registros en un momento del programa.
2. Durante una fase posterior de asignación a los registros, se escoge el registro específico en el que residirá una variable.

Es difícil encontrar una asignación óptima de registros a variables, incluso con valores en un solo registro. Matemáticamente, el problema es NP completo. Este problema se complica aún más el hardware, el sistema operativo o ambos, en la máquina objeto pueden exigir que se cumplan ciertas convenciones del uso de registros.

**Elección del orden de evaluación**: el orden en que se realizan los cálculos puede variar la eficiencia del código objeto. Algunos ordenamientos de los cálculos necesitan menos registros que otros para guardar resultados intermedios.

Elegir un orden mejor es un problema difícil, NP-completo.

LA MÁQUINA OBJETO

Se utilizará como computador objeto una máquina de registros representativa de varios minicomputadores: el lenguaje ensamblador.

COSTOS DE LAS INSTRUCCIONES

Los modos de direccionamiento junto con sus formas en lenguaje ensamblador y costos asociados son como sigue:

|  |  |  |  |
| --- | --- | --- | --- |
| MODO | FORMA | DIRECCIÓN | COSTO AÑADIDO |
| *Absoluto* | M | M | 1 |
| *registro* | R | R | 0 |
| *Indizado* | *c (R)* | *c*+ contenido(R) | 1 |
| *Registro indirecto* | \*R | contenido(R) | 0 |
| *Indizado indirecto* | \**c* (R) | contenido (*c* + contenido(R)) | 1 |

Se considera que el costo de una instrucción es uno más los costos asociados con los modos de dirección fuente y destino (que se indican como costo añadido en la tabla anterior). Este costo corresponde a la longitud (en palabras) de la instrucción. Los modos de direccionamiento que implican registros tienen costo cero, mientras que los de una posición de memoria o una literal tienen costo uno, porque dichos operandos tienen que almacenarse con la instrucción.

En la mayoría de las máquinas y de las instrucciones, el tiempo empleado en traer una instrucción de la memoria excede al tiempo empleado en ejecutar la instrucción. Por tanto, si se minimiza la longitud de las instrucciones también se tiende a minimizar el tiempo empleado en ejecutar las instrucciones.

Algunas de las dificultades de generar código para esta máquina se ven cuando se considera el código que se debe generar para una proposición de tres direcciones de la forma a := b + c. Esta proposición se puede implantar con muchas secuencias distintas de instrucciones. Ejemplos:

1. MOV b, R0

ADD c, R0 costo = 6

MOV R0, a

1. MOV b ,a

ADD c, a costo = 6

Suponiendo que R0, R1 y R2 contienen las direcciones de a, b y c, respectivamente, se puede utilizar:

1. MOV \*R1, \*R0

ADD \*R2, \*R0 costo = 2

Suponiendo que R1 y R2 contienen los valores de b y c, respectivamente, y que no se necesita el valor de b después de la asignación, se puede utilizar:

1. ADD R2, R1

MOV \*R1, a costo = 3

Se observa que para generar buen código para esta máquina, hay que utilizar sus capacidades eficientemente. Es preferible conservar el valor de lado izquierdo o de lado derecho de un nombre en un registro, si es posible, si se va a utilizar en un futuro cercano.

* + - 1. BLOQUES

Una representación de grafos de proposiciones de tres direcciones, llamado grafo de flujo, es útil para entender los algoritmos de generación de código. Los nodos del grafo de flujo representan cálculos y las aristas representan el flujo de control.

BLOQUES BÁSICOS

Un bloque básico es una secuencia de proposiciones consecutivas en las que el flujo de control entra al principio y sale al final sin detenerse y sin posibilidad de saltar excepto al final.

Ejemplo de bloque básico:

t1 := a \* a

t2 := a \* b

t3 := 2 \* t2

t4 := t1 + t3

t5 := b \* b

t6 := t4 + t5

Una proposición de tres direcciones *x* := *y* + *z* define *x* y usa (o se refiere a) *y* y *z*. Se dice que un nombre en un bloque básico está activo en un punto dado si su valor se utiliza después de ese punto en el programa, tal vez en otro bloque básico.

ALGORITMO: Partición en bloques básicos.

*Entrada*: una 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. Se determina el conjunto de líderes, la primera proposición de cada bloque básico.
   * La primera proposición es un líder.
   * Cualquier proposición que sea el destino de un salto condicional o incondicional es un líder.
   * Cualquier proposición que vaya inmediatamente después de un salto condicional o incondicional es un líder.
2. Para cada líder, su bloque básico consta del líder y de todas las proposiciones hasta, pero sin incluirlo, el siguiente líder o el fin del programa.

TRANSFORMACIONES EN BLOQUES BÁSICOS

Un bloque básico calcula un conjunto de expresiones. Estas expresiones son los valores de los nombres activos al salir del bloque. Se dice que dos bloques básicos son equivalentes si calculan el mismo conjunto de expresiones.

Se pueden aplicar varias transformaciones a un bloque básico sin modificar el conjunto de expresiones calculadas por el bloque. Existen dos tipos de transformaciones:

TRANSFORMACIONES QUE PRESERVAN LA ESTRUCTURA

Las principales transformaciones que preservan la estructura en bloques básicos son:

1. eliminación de subexpresiones comunes
2. eliminación de código inactivo
3. renombramiento de variables temporales
4. intercambio de dos proposiciones adyacentes independientes

Se supone que los bloques básicos no tienen matrices apuntadoras o llamadas a procedimientos.

1. *Eliminación de subexpresiones comunes*. Considérese el bloque básico

a := b + c

b := a – d

c := b + c

d := a – d

La segunda y la cuarta proposiciones calculan la misma expresión, b + c – d, y por tanto este bloque básico se puede transformar en el bloque equivalente

a := b + c

b := a – d

c := b + c

d := b

1. *Eliminación de código inactivo.* Supóngase que x esta inactivo en el punto en que la proposición x := y + z aparece en un bloque básico. Entonces esta proposición se puede eliminar sin modificar el valor del bloque básico.
2. *Renombramiento de variables temporales.* Siempre se puede transformar un bloque básico en un bloque equivalente en el que cada proposición que define un temporal define un temporal nuevo. Dicho bloque básico se denomina bloque en *forma normal*.
3. *Intercambio de proposiciones.* Si se tiene el siguiente bloque

t1 := b + c

t2 := x + y

Entonces se pueden intercambiar las dos proposiciones sin que esto afecte al valor del bloque si, y solo si, ni x ni y son t1 y ni b ni c son t2.

TRANSFORMACIONES ALGEBRAICAS

Las transformaciones útiles son las que simplifican las expresiones o sustituyen operaciones caras por otras más baratas. Por ejemplo, las proposiciones como

x := x + 0 ó

x := x \* 1

Se pueden eliminar de un bloque básico sin cambiar el conjunto de expresiones que calcula.

GRAFOS DE FLUJO

Los nodos del grafo de flujo son los bloques básicos. Un nodo se distingue como inicial; es el bloque cuyo líder es la primera proposición. Hay una arista dirigida del bloque B1 al bloque B2 si

1. hay un salto condicional o incondicional desde la última proposición de B1 a la primera proposición de B2, ó
2. B2 sigue inmediatamente a B1 en el orden del programa, y B1 no termina con un salto incondicional.

REPRESENTACIÓN DE BLOQUES BÁSICOS

Grafo de flujo para un programa:

La arista no indica si el salto condicional al final al final de *B* (si es que hay un salto condicional) va al líder de B’ cuando la condición se cumple o cuando no se cumple.

**Lazos**

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 lazo a cualquier otro, hay un camino de longitud uno o más, siempre dentro del lazo, y

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

Un lazo que no contiene otros lazos se denomina lazo interno.

**Informacion sobre el siguiente uso**

El uso de un nombre en una proposición de tres direcciones se define de la siguiente manera.Supóngase que la proposición de tres direcciones i asigna un valor a x. Si la proposición j tiene a x como un operando y el control puede fluir de la proposición i a la j por un camino que no tiene asignaciones intermedias a x, entonces se dice que la proposición j usa el valor de x calculado en i.

Se desea determinar para cada proposición de tres direcciones x: = y op z cuáles son lossiguientes usos de x, y y z.

El algoritmo que aquí se presenta para determinar los usos siguientes realiza una pasada hacia atrás sobre cada bloque básico. Se puede examinar fácilmente una cadena de proposiciones detres direcciones para encontrar los finales de los bloques básicos como en el algoritmo 9.1. Como los procedimientos pueden tener efectos secundarios arbitrarios, se supone por conveniencia que cada llamada a un procedimiento inicia un nuevo bloque básico.

Habiendo encontrado el final de un bloque básico, se inspecciona hacia atrás hasta el comienzo, registrando (en la tabla de símbolos) para cada nombre x si x tiene o no un siguiente uso en elbloque y si no lo tiene, indicando si está activo a la salida de ese bloque. Si se ha hecho el análisis del flujo de datos que se presenta en el capítulo 10, se sabe qué nombres están activos a la salida de cada bloque. Si no se ha hecho el análisis de variables activas, se puede suponer qué todas las variables no temporales están activas a la salida. Si los algoritmos que generan el código intermedio o que optimizan el código permiten que ciertos temporales se utilicen a través debloques, éstos también se deben considerar activos. Sería buena idea marcar dichos temporales,de modo que no haya que considerar activos todos los temporales.

Supóngase que se alcanza la proposición de tres direcciones i: x : = y op z en el examen haciaatrás. Entonces se hace lo siguiente:

1. Se asocia a la proposición i la información encontrada en la tabla de símbolos relativa al siguiente uso y actividad de x, y y z.

2. En la tabla de símbolos, se asigna a x "no activo" y "sin uso siguiente".

3. En la tabla de símbolos, se indica que y y z están "activos" y se igualan los siguientes usos y y z a i. Obsérvese que no se puede intercambiar el orden de los pasos 2 y 3 porque x puede ser y o z.

Si la proposición de tres direcciones i es de la forma x : = y o x : = op y, los pasos son los mismos que antes, sin tener en cuenta z.

UN GENERADOR DE CÓDIGO SIMPLE

Para cada operador dentro de una proposición hay un operador correspondiente en el lenguaje objeto.

Los resultados calculados se pueden dejar en registros mientras sea posible.

Para evitar un posible error, el algoritmo de generación de código simple guarda todo en memoria cuando se traslada a través de los límites de los bloques básicos así como cuando se hacen llamadas a procedimientos.

**D****escriptores de registros y direcciones**

1.-Un descriptor de registros sabe que lo que hay en cada registro. Es consultado siempre que se necesite un nuevo registro. Conforme avanza la generacion de codigo para el bloque, cada registro contiene siempre el valor de cero o mas nombres.

2.-Un descriptor de direcciones conoce la(s) posicion(es) donde se puede encontrar el valor en curso del nombre durante la ejecucion. La posicion puede ser un registro, una posicion en la pila, una direccion de memoria o un conjunto de estos porque cuando se copia, un valor tambien permanece donde estaba. Esta informacion se puede almacenar en la tabla de simbolos y se usa para determinar el metodo de acceso a un nombre.

ALGORITMO PARA GENERACIÓN DE CÓDIGO

Toma como entrada una secuencia de proposiciones de tres direcciones que constituyen un bloque básico. Para cada proposición de tres direcciones de la forma x := y op z se realizan las siguientes operaciones:

1. Se invoca la función *obtenreg* para determinar la posición L donde se debe guardar el resultado del calculo y op z. Generalmente L será un registro, pero también puede ser una posición de memoria. Dentro de poco se describirá obtenreg.
2. Se consulta el descriptor de direcciones de y para determinar y’, (una de) la(s) proposicion(es) en curso de y. Se prefiere el registro para y’ si el valor de y esta en ese momento en memoria y en un registro. Si el valor de y no esta todavia en L, se genera la instrucción MOV y’, L para colocar una copia de y en L.
3. Se genera la instrucción OP z’, L donde z’ es una posición en curso de x. De nuevo se prefiere un registro a una posición en memoria si z esta en ambos. Se actualiza el descriptor de direcciones de x para indicar que x esta en la posición L. Si L es un registro se actualiza su descriptor para indicar que contiene el valor de x, y se elimina x de todos los otros descriptoresde registros.
4. Si los valores en curso de y o z, o ambos, no tienen usos siguientes, no están activos a la salida del bloque, y están en registros, se altera el descriptor de registros para indicar que después de la ejecución de x := y op z, estos registros ya no contendrán y o z, o ambos, respectivamente.

La función *obtenreg*

La función oblenreg devuelve la posición L para guardar el valor de x para la asignación x : = y op x.

1. Si el nombre y esta en un registro que no contiene el valor de otros nombres (recuérdese que las instrucciones de copia como x : = y podrían hacer que un registro guardara el valor de dos o más variables simultáneamente), e y no esta activa y no tiene uso siguiente después de la ejecución de x := y op z, entonces se devuelve el registro de y para L. Se actualiza el descriptor de direcciones, de y para indicar que y ya no se encuentra en L
2. Si falla (1), devuélvase un registro vació para L si hay alguno.
3. Si falla (2), si x tiene un uso siguiente en el bloque, u op es un operador, como indizar, que exige un registro, encuéntrese un registro ocupado R. Almacénese el valor de R en una posición de memoria (mediante MOV R, M) si es que todavía no está en una posición de memoria apropiada M, actualícese el descriptor de direcciones de M y devuélvase R. Si R contiene el valorde varias variables, se debe generar una instrucción MOV para cada variable que haya que almacenar. Un registro ocupado adecuado puede ser uno cuyo dato sea el referenciado más lejos en el futuro, o uno cuyo valor también esté en memoria. No se especifica la opción exacta porque no se conoce mejor forma de hacer la selección.
4. Si no se utiliza x en el bloque, o no se puede encontrar ningún registro ocupado adecuado, selecciónese la posición de memoria de x para L.

Ejemplo 9.5. La asignación d : = (a-b) + (a-c) + (a-c) se puede traducir a la ¡siguiente secuencia de código de tres direcciones:

t := a - b

u := a - c

v := t + u

d := v + u

con d activa al final.

|  |  |  |  |
| --- | --- | --- | --- |
| Proposiciones | Codigo generado | descriptor de registros | descriptor de direcciones |
|  |  | Registros vacios |  |
| t := a - b | MOV a, R0  SUB b, R0 | R0 contiene a t | t en R0 |
| u := a - c | MOV a, R1  SUB c, R1 | R0 contiene a t  R1 contiene a u | t en R0  u en R1 |
| v := t + u | ADD R1, R0 | R0 contiene a v  R1 contiene a u | u en R1  v en R0 |
| d := v + u | ADD R1, R0  MOV R0, d | R contiene a d | d en R0  d en R0 y  en memoria |

GENERACIÓN DE CÓDIGO PARA OTRO TIPO DE PROPOSICIONES

Secuencias de código para asignaciones con índices:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PROPOSICIÓN | i EN EL REGISTRO Ri | | | i EN MEMORIA Mi | | | i EN LA PILA | |
| CÓDIGO | COSTO | CÓDIGO | | COSTO | CÓDIGO | | COSTO |
| a := b [ i ] | MOV b(Ri), R | 2 | MOV Mi, R  MOV b(R),R | | 4 | MOV Si(A),R  MOV b(R),R | | 4 |
| a [ i ] := b | MOV b,a(Ri) | 3 | MOV Mi, R  MOV b,a(Ri) | | 5 | MOV Si(A),R  MOV b,a(Ri) | | 5 |

Secuencias de código para asignaciones con apuntadores:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PROPOSICIÓN | p EN EL REGISTRO Rp | | | p EN MEMORIA Mp | | | p EN LA PILA | |
| CÓDIGO | COSTO | CÓDIGO | | COSTO | CÓDIGO | | COSTO |
| a := \* p | MOV \*Rp, a | 2 | MOV Mp, R  MOV \*R,R | | 3 | MOV Sp(A),R  MOV \*R, R | | 3 |
| \* p := a | MOV a, \*Rp | 2 | MOV Mp, R  MOV a, \*R | | 4 | MOV a,R  MOV R,\*Sp(A) | | 4 |

Proposiciones condicionales:

Las máquinas implantan los saltos condicionales en una de dos formas. Una manera es saltar si el valor de un registro designado cumple una de seis condiciones: negativo, cero, positivo, no negativo, no cero y no positivo. En dicha máquina se puede implantar una proposición de tres direcciones como if x<y goto z restando y a x en el registro R, y saltando después a z si el valor del registro R es negativo.

Un segundo enfoque utiliza un conjunto de códigos de condición para indicar si a última cantidad calculada o cargada en un registro es negativa, cero o positiva.

A menudo una instrucción de comparación (CMP en la máquina de este capítulo) tiene la ventajade que asigna el código de condición sin calcular un valor. Es decir, CMP x, y pone el código de condición en positivo si x>y, y así sucesivamente. Una instrucción de máquina de salto condicional realiza el salto si se cumple una condición designada <, =, >, <=, !=, o >=. Se utiliza la instrucción CJ < = z para indicar que "salta a z si el código de condición es negativo o cero". Porejemplo, ir x<y goto z se podría implantar mediante

CMP x, y

CJ< z

Si se está generando código para una máquina con códigos de condición es útil mantener un descriptor de códigos de condiciones cuando se genera el código. Este descriptor indica el último nombre en asignar el código de condición, o el par de nombres comparados, si el código decondición se asignó por última vez de esta forma. Por tanto, se podría implantar

x := y + z,

if x < 0 goto z

mediante

MOV y, R0

ADD z, R0

MOV R0, x

CJ< z

si se supiera que el código de condición fue determinado por x después de ADD z, R0.

**DISTRIBUCION Y ASIGNACION DE REGISTROS**

Las instrucciones que implican sólo operandos en registros son más cortas y rapida que las de operandos en memoria. Por tanto, es importante utilizar eficientemente los registros para generar buen código.

Un aspecto de la distribución y asignación de los registros es asignar valores específicos en un programa objeto a algunos registros. Por ejemplo, se puede tomar la decisión de asignar las direcciones base a un grupo de registros, los cálculos aritmeticos a otro, el tope de la pila de ejecución a un registro fijo, y así sucesivamente.

Este enfoque tiene la ventaja de que simplifica el diseño de un compilador. El inconveniente esque, si se aplica con demasiada rigidez, no utiliza eficientemente los registros; algunos registros pueden quedar sin utilizar durante partes substanciales de código, mientras que se generan cargas y almacenamientos innecesarios. Sin embargo, en la mayoría de los entornos de cálculo es razonable reservar algunos registros para los registros base, apuntadores a la pila, etc., y permitir que el compilador utilice los restantes registros como crea conveniente.

**Distribución de registros globales**

El algoritmo de generación de código de la sección 9.6 utilizaba registros para guardar valores en el tiempo de duración de un bloque básico. Sin embargo, todas las variables activas se almacenaban en memoria al final de cada bloque. Para evitar algunos de estos almacenamientos y cargas correspondientes, se puede decidir asignar los registros a las variables utilizadas con frecuencia y mantener consistentes estos registros a través de los límites de los bloques (globalmente). Como los programas pasan la mayor parte de su tiempo en lazos internos, un enfoque natural de la distribución global de registros es intentar conservar un valor utilizado con frecuencia en un registro fijo durante todo un lazo. Por el momento, supóngase que se-conoce la estructura de lazo de un grafo de flujo y los valores calculados en un bloque básico que seutilizan fuera de ese bloque. El siguiente capítulo estudia las técnicas para calcular esta información.

Una estrategia para la distribución global de registros es asignar un número fijo de registros para guardar los valores más activos de cada lazo interno. Los valores seleccionados pueden ser distintos en lazos distintos. Se pueden utilizar los registros que todavía no se hayan asignado para conservar los valores locales a un bloque, como en la sección 9.6. Este enfoque tiene el defecto de que el número fijo de registros no siempre es el número correcto de registros disponibles parala asignación global de registros. Sin embargo, el método es sencillo de implantar y se ha usado en FORTRAN H, el compilador optimador de FORTRAN para las máquinas de la serie IBM-360(Lowry y MedIock [19691).

En lenguajes como C y BLISS, un programador puede realizar una asignación de registros directamente utilizando declaraciones de registros para conservar algunos valores dentro de los registros para la duración de un procedimiento. Un uso inteligente de las declaraciones de registros puede acelerar muchos programas, pero un programador no debe realizar la asignación de los registros sin perfilar primero su programa.

**Cuentas de uso**

Un método sencillo para determinar los ahorros obtenidos al conservar la variable x en unregistro durante el tiempo del lazo L es reconocer que en este modelo de máquina se ahorra una unidad de costo por cada referencia a x si x está en un registro. Sin embargo, si se utiliza el enfoque de la sección anterior para generar código para un bloque, es muy probable que después de haber calculado x en un bloque permanecerá en un registro si hay usos posteriores de x en dicho bloque. Por tanto, se contabiliza un ahorro de uno por cada uso de x dentro del lazo L que no venga precedido de una asignación a x en el mismo bloque. Así, si se asigna un registro a x, se contabiliza un ahorro de dos por cada bloque de L para el que x esté activo a la salida y en el que a x se le asigna un valor.

Por parte del débito, si x está activa en la entrada al encabezamiento del lazo, se debe cargar x en un registro justo antes de entrar en el lazo L. Esta carga cuesta dos unidades. De manera similar, para cada salida de un bloque B del lazo L en el x esté activo en la entrada a algún sucesor de B fuera de L, se debe almacenar memoria x con un costo de dos. Sin embargo, en el supuesto de que el lazo se repita muchas veces, se pueden ignorar estos débitos porque ocurren sólo una vez cuando se entra al lazo. Por tanto, una fórmula aproximada para la ventaja obtenida al asignar un registro a x dentro del lazo L es:

∑ ( uso (x, B) + 2\*activo (x, B ))

bloques B en L

donde uso(x,B) es el número de veces que se utiliza x en B antes de cualquier definición de x; activo(x,B) es 1 si x está activo a la salida de B y se le asigna un valor en B, y en caso contrarioes 0. Obsérvese que (9.4) es aproximada, porque no los bloques en un lazo se ejecutan con la misma frecuencia y también porque se basaba en el supuesto de que un lazo se repite "muchas" veces. En otras maquinas, habrá que desarrollar una fórmula análoga a (9.4), aunque posiblemente bastante distinta de ella.

Ejemplo 9.6.

Considérense los bloques básicos dentro del lazo interno representado en la figura 9.13, donde se han omitido proposiciones de salto y salto condicional. Supóngase que se asignan los registros R0, Rl y R2 para guardar valores a traves del lazo. Las variables activas a la entrada y a la salida de cada bloque se muestran por conveniencia en la figura 9.13, inmediatamente por encima y por debajo de cada bloque, respectivamente. En el capítulo 10 se señalan algunos puntos sutiles sobre las variables activas. Por ejemplo, obsérvese que tanto e como f están activos al final de B1, pero sólo e está activo a la entrada de B2 Y Sólo f está activo a la entrada de B3. En general, las variables activas al final de un bloque son la unión de las variables activas al comienzo de cada uno de sus bloques sucesores.

Para evaluar (9.4) para x = a se observa que a está activa a la salida de B1 y se le asigna un valor allí, pero no está activa a la salida de B2, B3 o B4. Por tanto, ∑ 2 \* activo (a,B) = 2. Asimismo, uso (a, B1) = 0, porque a se define en B1 antes

de cualquier uso. También, uso (a, B2) = uso (a, B3) = 1 y uso (a, B4) = 0. Por tanto, ∑ uso (a,B) = 2. Así, el valor de (9.4) para x = a es 4. Es decir, se pueden ahorrar cuatro unidades de costo seleccionando a para uno de los registros globales. Los valores de (9.4) para b, c, d, e y f son 6, 3, 6, 4 y 4, respectivamente. De ese modo se pueden seleccionar a, b y d para los registros R0, Rl y R2, respectivamente. Utilizar para e o f en lugar de para a sería otraopción con la misma ventaja aparente. En la figura 9.14 se muestra el código ensamblador generado a partir de la figura 9.13, suponiendo que se utiliza la estrategia de la sección 9.6 para generar código para cada bloque. No se muestra el código generado para los saltos condicionales o incondicionales que finalizan cada bloque de la figura 9.13, y por tanto no se muestra el código generado como secuencia simple,como aparecería en la práctica. Se hace observar que, si no se ciñe uno a la estrategia de reservar R0, Rl y R2, se podría utilizar

SUB R2, R0

MOV R0, f

para B2, ahorrando una unidad ya que no esta activo a la salida de B2. Se podria realizar en B3 un ahorro similar.

**Asignación a registros para lazos externos**

Habiendo asignado registros y generado código para los lazos internos, se puede aplicar la misma idea para lazos cada vez más grandes. Si un lazo externo L1 contiene un lazo interno L2, no hay que asignar registros en L1 - L2 a los nombres a los que se les asignaron registros en L2, Sin embargo, si al nombre x se le asignó un registro en el lazo L1 pero no en L2, se debe almacenar x a la entrada de L2 y cargar x si se abandona L2 y se entra a un bloque de L1 - L2. De manera similar, si se elige asignar a x un registro en L2 pero no en L1 se debe cargar x a la entrada de L2 y almacenar x en memoria a la salida de L2.

**Asignación de los registros mediante coloración de grafos**

Cuando se necesita un registro para un cálculo pero todos los registros disponibles están siendo utilizados, se debe almacenar (vaciar) el contenido de uno de los registros utilizados en una posición de memoria para dejar libre un registro. La coloración de grafos es una técnica sistemática y sencilla para asignar registros y administrar el vaciado de los registros.

En este método se utilizan dos pasadas. En la primera, se seleccionan instrucciones de la máquina objeto como si hubiera un número infinito de registros simbólicos; en efecto, los nombres utilizados en el código intermedio se convierten en nombres de registros y las proposiciones de tres direcciones se convierten en proposiciones, en el lenguaje de máquina. Si el acceso a las variables exige instrucciones que utilicen apuntadores a la pila, apuntadores al display, registros de base u otras cantidades que faciliten el acceso, entonces se supone que estas cantidades se conservan en registros reservados para cada propósito. Normalmente su uso es directamente traducible a un modo de acceso para una dirección mencionada en una instrucción de máquina. Si el acceso es más complejo, éste se puede dividir en varias instrucciones de máquina, y puede ser necesario crear un registro simbólico temporal (o varios).

Una vez seleccionadas las instrucciones, una segunda pasada asigna registros fisicos a los registros simbólicos. El objetivo es encontrar una asignación que minimice el costo de los vaciados de registros.

En la segunda pasada se construye para cada procedimiento un grafo de interferencia entre registros en el que los nodos son registros simbólicos y una arista conecta dos nodos si uno está activo en un punto donde se define el otro. Por ejemplo. un grafo de interferencia entre registros para la figura 9.13 tendría nodos para los, nombres a y d. En el bloque B1, a está activa en la segunda proposición, que define a d; por tanto, en el grafo habría una arista entre los nodos de a y d.

Se intenta colorear el grafo de interferencia entre registros utilizando k colores donde k es el número de registros asignables. (Se dice que un grafo está coloreado si a cada nodo se le ha asignado un color de modo que no haya dos nodos adyacentes con el mismo color.) Un color representa un registro y la coloración garantiza que no se asigna el mismo registro físico a dos registros simbólicos que pueden interferir uno con el otro.

Aunque el problema de determinar si un grafo es coloreable es generalmente es NP- completo se puede utilizar la siguiente técnica heurística en la práctica para realizar el coloreado rápidamente. Supóngase que un nodo n en un grafo G tiene menos que k vecinos (nodos conectados a n por una arista). Elimínese n y sus aristas de G para obtener un grafo G'. Un k-coloreado de G' se puede ampliar a un k-coloreado de G asignando a n un color no asignado a ninguno de sus vecinos.

Eliminando repetidamente el grafo de interferencias entre registros los nodos con menos de k aristas, se obtiene el grafo vacío, en cuyo caso se puede producir un k-coloreado del grafo original coloreando los nodos en orden inverso al que fueron eliminados o bien se obtiene un grafo en el que cada nodo tiene k o más nodos adyacentes. En este último caso, ya no es posible un k-coloreado. Llegados a este punto, un nodo se vacía introduciendo código para almacenar en memoria y recargar el registro. Entonces se modifica adecuadamente el grafo de interferencias y se continúa el proceso de coloreado. Como regla general se evita introducir código para vaciar registros en lazos internos.

Libro del dragón Capitulo 10

**Criterios para las transformaciones para mejorar el código**

Dicho de una manera sencilla, las mejores transformaciones de programas son las que producen elmayor beneficio con el menor esfuerzo. Las transformaciones utilizadas por un compilador optimador deben tener varias propiedades.

Primero, una transformación debe preservar el significado de los programas. Es decir, una"optimación" no debe cambiar el resultado producido por un programa para una entrada dada, o causar un error, como una división por cero, que no estuviera presente en la versión original del programa fuente. La influencia de este criterio impregna este capítulo. En todo momento se toma el enfoque "seguro de desaprovechar la oportunidad de aplicar una transformación en lugar de cambiar lo que hace el programa.

Segundo, una transformación debe, como promedio, acelerar los programas en una cantidad mensurable. En ocasiones interesa reducir el espacio que ocupa el código compilado, aunque eltamaño del código tiene menos importancia que la que tenía antes. Por supuesto, no toda transformación consigue mejorar todo programa y, ocasionalmente, una "optimación" puede ralentizar ligeramente un programa, mientras en general mejore las cosas.

Tercero, una transformación debe valer la pena. No tiene sentido que el escritor de un compilador haga el esfuerzo intelectual de aplicar una transformación que mejore el código y que el compilador gaste el tiempo adicional compilando programas fuente si este esfuerzo no es recompensado cuando se ejecutan los programas objeto. Algunas transformaciones locales o "de mirilla" del tipo estudiado en la sección 9.9 son lo bastante sencillas y ventajosas como para ser incluidas en un compilador.

Algunas transformaciones sólo se pueden aplicar después de un análisis detallado y que lleva su tiempo del programa fuente, de modo que tiene poco sentido aplicarlas a programas que sólo se ejecutarán pocas veces. Por ejemplo, es probable que un compilador no optimador rápido sea más útil durante la depuración o para "trabajos de estudiantes" que se ejecutarán con éxito pocas veces y luego se desecharán. Sólo cuando el programa en cuestión toma una fracción significativa de los cíclos de la máquina, la calidad mejorada del código justifica el tiempo empleado en ejecutar un compilador optimador sobre el programa.

**Obtención de un mayor rendimiento**

Generalmente se obtienen mejoras espectaculares en el tiempo de ejecución de un programa -como reducir el tiempo de ejecución de unas horas a unos segundos mejorando el programa a todos los niveles, desde el nivel fuente hasta el nivel objeto, como se sugiere en la figura 10. 1. En cada nivel, las opciones disponibles están entre los dos extremos de encontrar un algoritmo mejor y de implantar un algoritmo dado, así que se realizan menos operaciones.

Las transformaciones de los algoritmos producen ocasionalmente mejoras espectaculares en el tiempo de ejecución. Por ejemplo, Bentley 119821 dice que el tiempo de ejecución de un programa para clasificar N elementos se redujo de 2.02N2 Microsegundos a 12log2N microsegundos cuando una "clasificación por inserción" se susituyó por una "clasificación por particiones" (quicksort) 1. Para N = 100, la sustitución acelera el programa en un factor de 2.5.Para N = 100 000, la mejora es mucho mayor: la sustitución acelera el programa en un factor de más de mil.

Lamentablemente, ningún compilador puede encontrar el mejor algoritmo para un problema dado. Sin embargo, a veces un compilador puede sustituir una secuencia de operaciones por una secuencia algebraicamente equivalente, y con ello reducir significativamente el tiempo de ejecución de un programa. Dichos ahorros son más habituales cuando se aplican transformaciones algebraicas a los programas de lenguajes de muy alto nivel; por ejemplo,lenguajes de consulta para bases de datos (véase Ullman [ 1982]).

En esta sección y en la siguiente, se utilizará un programa de clasificación llamado clasificación por particiones para ilustrar el efecto de varias transformaciones para mejorar el código. El programa en C de la figura 10.2 viene de Sedgewick [1978]\_ donde se analiza la optimación a mano de dicho programa. Aquí no se analizarán. los aspectos algorítmicos de este programa -de hecho, a [0] debe contener el menor y a[max] el mayor elemento por clasificarse para que elprograma funcione

void clasificación-por-particiones(int m, int n){

int ¡,j;

int v,j;

if ( n <= ) return;

/\* elfragmento comienza aquí\*/

i = m-1; j = n; v = a[n];

while(1)

do i = i+1; while (a[i] < v);

do j = j-1; while (a[j] > v);

if( i >= j ) break;

x = a[i]; a[i] = a[j]; a[j] = x;

}

x = a[i]; a[i] = a[n]; a[n] = x;

/\* el fragmento termina aquí\*/

clasificación\_por\_particiones(m, j);

clasificación\_por\_particiones(i+l, n);

Código en C para clasificación-por-particiones.

Puede que sea posible realizar algunas transformaciones de mejora del código a nivel del programa fuente. Por ejemplo, en un lenguaje como Pascal o FORTRAN. un programador sólo puede hacer referencia a elementos de una matriz de la manera habitual; por ejemplo, como b [ i, j ]. En el nivel del lenguaje intermedio, sin embargo, pueden aparecer nuevas oportunidades para mejorar el código. El código de tres direcciones, por ejemplo, proporciona muchas oportunidades para mejorar los cálculos de las direcciones, especialmente en lazos. Considérese el código de tres direcciones para determinar el valor de a[i], suponiendo que cada elemento de la matriz ocupa cuatro bytes:

t1 := 4\*i; t2 := a[t1]

Un código intermedio ingenuo recalculará 4\*i cada vez que a[i] aparezca en el programa fuente, y el programador no controla los cálculos redundantes de las direcciones porque están implícitas en la implantación del lenguaje, en lugar de ser explícitas en el código escrito por el usuario. En estas situaciones, corresponde al compilador depurarlas. Sin embargo, en un lenguaje como C, esta transformación puede ser realizada al nivel fuente por el programador, ya que las referencias a los elementos de una matriz se pueden reescribir sistemátícamente utilizando apuntadores para hacerlas más eficientes. Esta reescritura es similar a las transformaciones que tradicionalmente aplican los compiladores optimadores de FORTRAN.

En el nivel de la máquina objeto, corresponde al compilador utilizar adecuadamente los recursos de la máquina. Por ejemplo, conservar las variables más utilizadas dentro de registros puede reducir significativamente el tiempo incluso a la mitad. De nuevo, C permite a un programador aconsejar al compilador que conserve algunas variables en registros, pero la mayoría de los lenguajes no lo permite. De manera similar, el compilador puede acelerar los programas de forma significativa eligiendo instrucciones que aprovechan los modos de direccionamiento de la máquina para hacer en una instrucción lo que ingenuamente se supondría que necesitaran dos o tres, como se estudió en el capítulo 9. Aunque es posible que el programador mejore el código, puede ser más conveniente que el compilador realice algunas de las mejoras. Si se puede confiar en que el compílador genere código eficiente, entonces el usuario puede concentrarse en escribir código claro.

**Una organizacíón para un compfiador optimador**

Como ya se ha mencionado, existen varios niveles en los que se puede mejorar un programa. Como las técnicas necesarias para analizar y transformar un programa no cambian significativamente con el nivel, este capítulo se concentra en las transformaciones de código intermedio utilizando la organización que se muestra en la figura 10.3. La fase de mejora del código consta del análisis de flujo de control y el análisis de flujo de datos seguidos de la aplicación de transformaciones. El generador de código, que se estudia en el capítulo 9, produce el programa objeto a partir del código intermedio transformado.

Para una mejor presentación, se supone que el código intermedio consta de proposiciones de tres direcciones. El código intermedio, del tipo producido por las técnicas del capítulo 8, para una parte del programa de la figura 10.2, se muestra en la figura 10.4. Con otras representaciones intermedias, las variables temporales tl, t2, ..., t15 de la figura 10.4 no tienen que aparecer explícitamente, como se estudia en el capítulo 8

La organización de la figura 10.3 tiene las siguientes ventajas:

1 . Las operaciones necesarias para implantar construcciones de alto nivel se hacen explícitas en el código intermedio, de modo que es posible optimarlas. Por ejemplo, los cálculos de direcciones para a [ i ] se hacen explícitos en la figura 10.4, de modo que el recálculo de expresiones como 4\*i se puede eliminar como se estudia en la siguiente sección.

2. El código intermedio puede ser (relativamente) independiente de la máquina objeto, de modo que el optimador no tiene que cambiar mucho si el generador de código se sustituye por uno para una máquina diferente. El código intermedio de la figura 10.4 asume que cada elemento de la matriz a ocupa cuatro bytes. Algunos códigos intermedios, como el código P para Pascal, dejan al generador de código que proporcione el tamaño de los elementos de las matrices, así que el código intermedio es independiente del tamaño de la palabra de una máquina. Se podría haber hecho lo mismo en este código intermedio si se sustituyera 4 por una constante simbólica.

En el optimador de código, los programas se representan mediante grafos de flujo, en los que las aristas indican el flujo del control y los nodos representan bloques básicos, como se vio en la sección 9.4. A menos que se especifique lo contrario, un programa significa un solo procedimiento. En la sección 10.8 se analiza la optimación entre procedimientos.

Ejemplo 10.1. La figura 10.5 contiene el grafo de flujo para el programa de la figura 10.4. B1 es elnodo inicial. Todos los saltos condicionales e incondicionales hacia proposiciones en la figura10.4 se han sustituido en la figura 10.5 por saltos al bloque del cual son líderes las proposiciones.

En la figura 10.5 hay tres lazos, B2 y B3 son lazos en sí mismos. Los bloques B2, B3, B4 y B5 juntos forman lazo, con entrada B2.

**LAS PRINCIPALES FUENTES PARA LA OPTIMACION**

En esta sección se presentan algunas de las transformaciones más útiles para mejorar el código. Las técnicas para aplicar estas transformaciones se presentan en Posteriores secciones. Una transformación de un programa se denomina local si se puede realizar observando sólo lasproposiciones de un bloque básico; en caso contrario se denomina global. Muchastransformaciones se pueden realizar tanto a nivel local como global. Generalmente, primero se realizan las transformaciones locales.

**Transformaciones que preservan la función**

Hay varias formas en que un compilador puede mejorar un programa sin modificar la función que calcula. La eliminación de subexpresiones comunes, la programación de copias, la eliminación de código inactivo y el cálculo previo de constantes son ejemplos comunes de dichas transformaciones que preservan la función.

La sección 9.8 sobre representación por medio de GDA para los bloques básicos mostro cómo se pueden eliminar las subexpresiones comunes locales conforme se construye el GDA para el bloque básico. Las otras transformaciones aparecen principalmente cuando se realizan optimaciones globales, y se estudiarán una a una.

A menudo un programa incluira varios calculos del mismo valor, como un desplazamiento en una matriz. Como se mencionó en la sección 10. 1, el programador no puede evitar algunos de estos cálculos duplicados porque están bajo el nivel detalle accesible dentro del lenguaje fuente, Porejemplo, el bloque B5 que se muestra en la figura 10.6(a) recalcula 4 \* i y 4 \* j.

**Subexpresiones comunes**

Una ocurrencia de una expresión E se denomina subexpresión común si E ha sido previamente calculada y los valores de las variables dentro de E no han cambiado desde el cálculo anterior. Se puede evitar recalcular la expresión si se puede utilizar el valor calculado previamente. Por ejemplo, las asignaciones a t7 y a t10 tienen las subexpresiones comunes 4 \* i y 4 \* j, respectivamente, en el lado derecho de la figura 10.6(a). Han sido eliminados en la figura 10.6(b) utilizando t6 en lugar de t7 Y t8 en lugar de t10. Se obtendría este cambio si se reconstruyera el código intermedio a partir del GDA para el bloque básico.

Ejemplo 10.2. En la figura 10.7 se muestra el resultado de eliminar subexpresiones comunes globales y locales de los bloques B5 y B6 en el grafo de flujo de la figura 10.5. Primero se analiza la transformación de B5 y después se mencionan algunas sutilezas relativas a matrices.

Después de haber eliminado las subexpresiones comunes, B5 todavía tiene que evaluar 4 \* i y 4 \* j, como se muestra en la figura 10.6(b). Ambas son subexpresiones comunes; en particular, las tres proposiciones

t8 := 4\*j; t9 := a [t8] ; a[t8] := x

en B5 se pueden sustituir por

t9 : = a [t4]; a[t4] = x

utilizando t4 calculado en el bloque B3. En la figura 10.7, obsérvese que a medida que el controlpasa desde la evaluación de 4 \* j en B3 hasta B5, no hay cambio en j, de modo que se puede utilizar t4 Si se necesita 4 \* j.

Otra subexpresión común aparece en B5 después de que t4 sustituye a t8 nueva expresión a[t4] corresponde al valor de a [ j ] en el nivel fuente. No sólo j mantiene su valor a medida que elcontrol sale de B3 y después entra a B5, sino que también lo mantiene a [ j ], un valor calculado dentro de un temporal t5, porque no hay asignaciones a elementos de la matriz a en el ínterin. Las proposiciones

t9 := a[t4]; a[t6] := t9

en B5 se pueden por tanto sustituir por

a[t6] := t5

De manera análoga, el valor asignado a x en el bloque B5 de la figura l0.6(b) el mismo que elvalor asignado a t3 en el bloque B2. El bloque B5 de la figura 10.7, es el resultado de eliminar las subexpresiones comunes correspondientes a los valores de las expresiones del nivel fuente a [i] y a [j] de B5 en la figura 10. 6(b). En la figura 10.7 se han realizado una serie de transformaciones similares con B6.

La expresión a[t1] en los bloques B1 y B6 de la figura 10.7 no se considera una subexpresíón común, aunque t1 se puede utilizar en ambos lugares. Después de que el control abandona B1 y antes de que llega a B6 puede pasar por B5, donde hay asignaciones a a. Por tanto. a[t1] puede no tener el mismo valor al llegar a B6 que el que tenía al salir de B1, y no es seguro considerar a[t1] como una subexpresion común.

**Propagación de copias**

El bloque B5 de la figura 10.7 se puede mejorar aún más eliminando x por medio de dos nuevas transformaciones. Una concierne las asignaciones de la forma f := g llamadas proposiciones de copia, o copias simplemente. Si se estudia más detalladamente el ejemplo 10.2, las copias habrían surgido mucho antes porque el algoritmo para eliminar las subexpresiones comunes las introduce, al igual que otros algoritmos. Por ejemplo, cuando en la figura 10. 8 se elimina la subexpresión comun

en c: =d+e, el algoritmo utiliza una nueva variable t para guardar el valor de d+e Como el control puede llegar a c: =d+e, ya sea después de hacer la asignación a a o después de hacer la asignacióna b, sería incorrecto sustituir c:=d+e por c: =a o por c :=b.

La idea en que se basa la transformación de propagación de copias es utilizar g por f, siempre quesea posible después de la proposición de copia f :=g. Por ejemplo, la asignación x:=t3 en elbloque B5 de la figura 10.7 es una copia. La propagación de copias aplicada a B5 produce:

x := t3

a[t2] := t5

a[t4] := t3

goto B2

Esto puede no parecer una mejora, pero como ya se verá, da la oportunidad de eliminar las asignaciones a x.

**Eliminacion de codigo inactivo**

Una variable está activa en un punto de un programa si su valor puede ser utilizado posteriormente; en caso contrario está inactiva en ese punto.Una idea afin es el código inactivo oinútil, proposiciones que calculan valores que nunca llegan a utilizarse. Aunque es improbable que el programador introduzca código inactivo intencionadamente, puede aparecer como resultado de transformaciones anteriores. Por ejemplo, en la sección 9.9 se estudió el uso de depura que se asigna a falso o verdadero en varios puntos del programa, y se utiliza en proposiciones como

if (depura) print . . .

Mediante un análisis del flujo de datos, es posible concluir que cada vez que el programa alcanza dicha proposición, el valor de depura es falso. Generalmente, lo es porque hay una proposición determinada

depura := false

que se puede considerar la última asignación a depura antes de hacer la comprobación independientemente de la secuencia de ramificaciones que tome en realidad el programa. Si la propagación de copias sustituye depura por el valor de verdad false, entonces la proposición deimpresión print está inactiva porque no se puede alcanzar. Se puede eliminar la prueba y la impresión del código objeto. Generalmente, deducir en el momento de la compilación que elvalor de una expresión es una constante y utilizar la constante en su lugar se conoce como cálculo previo de constantes.

Una ventaja de la propagación de copias es que a menudo convierte la proposición de copia en código inactivo. Por ejemplo, la propagación de copias seguida de la eliminación de código inactivo elimina la asignación a x y transforma (10. 1) en:

a[t2] := t5

a[t4] := t3

goto B2

Este código es una mejora adicional del bloque B5 de la figura 10.7.

**Optimaciones de lazos**

Ahora se da una breve introducción a un lugar muy importante para las optimaciones, que son los lazos, especialmente los lazos internos donde los programas tienden a emplear la mayor parte de su tiempo, El tiempo de ejecución de un programa se puede mejorar sí se disminuye la cantidad de instrucciones en un lazo interno, incluso si se incrementa la cantidad de código fuera del lazo.

Hay tres técnicas importantes para la optimacíón de lazos: el traslado de código, que traslada código fuera del lazo; la eliminación de variables de inducción, que se aplica para eliminar i y j de los lazos internos B2 y B3 de la figura 10.7; y la reducción de intensidad, que sustituye una operación cara por una más barata, como una multiplicación por una suma.

**Traslado de código**

Una modificación importante que disminuye la cantidad de código en un lazo es el traslado de código. Esta transformación toma una expresión que produce el mismo resultado independientemente del número de veces que se ejecute un lazo (el cálculo de una invariante del ciclo) y coloca la expresión antes del lazo. Obsérvese que la noción "antes del lazo" supone la existencia de una entrada al lazo. Por ejemplo. la evaluación de limite - 2 es un cálculo de una invariante de lazo en la siguiente proposición while:

while ( i <= límite-2) /\* la proposición no cambia limite \*/

El traslado de código resultará equivalente a

t = límite-2;

while ( i <= t ) /\* la proposición no cambia límite ni t \*/

**Variables de inducción y reducción de intensidad**

Aunque el traslado de código no es aplicable al ejemplo de la clasificación por particiones que se ha venido considerando, las otras dos transformaciones sí lo son. Los lazos se procesan generalmente de dentro hacia afuera. Por ejemplo, considérese el lazo alrededor de B3. Sólo semuestra en la figura 10.9 la parte del grafo de flujo relevante para la transformación de B3

Obsérvese que los valores de j y t4 permanecen atados; cada vez que el valor de j disminuye en1, el de t4 disminuye en 4 porque 4 \* j se asigna a t4. Dichos identificadores se denominan variables de inducción.

Cuando hay dos o más variables de inducción en un lazo, es posible suprimirlos todos menos uno, mediante el proceso de eliminación de variables de inducción. Para el lazo interno alrededor de B3 en la figura 10.9(a), no se puede suprimir completamente j o t4; t4 se utiliza en B3 y j en B4. Sin embargo, se puede ilustrar la reducción de intensidad y una parte del proceso de eliminación de variables de inducción. Finalmente, j será eliminado cuando se considere el lazo externo de B2 - B5.

Ejemplo 10.3. Como la relación t4 = 4 \* j se cumple siempre después de dicha asignación a t4 en la figura 10.9(a) y t4 no cambia en ninguna otra parte del lazo interno alrededor de B3, se deduce que justo después de la proposición j : = j - 1 se debe cumplir la relación t4 = 4\*j-4. Por tanto, sepuede sustituir la asignación t4 := 4 \* j por t4: =t4- 4. El único problema es que t4 no tiene un valor cuando se entra al bloque B3 por primera vez. Como se debe conservar la relación t4 = 4 \* j a la entrada al bloque B3, se coloca una inicialización de t4 al final del bloque en donde se inicializa j misma, mostrada por la suma punteada al bloque B1 en la figura 10.9(b).

La sustitución de una multiplicación por una resta acelerará el código objeto si la multiplicación emplea más tiempo que la suma o la resta, como es el caso en muchas máquinas.

La sección 10.7 analiza cómo detectar las variables de inducción y las transformaciones que sepueden aplicar. Esta sección concluye con un ejemplo más de eliminación de variables de inducción que considera i y j en el contexto del lazo exterior que contiene B2, B3, B4 y B5

Ejemplo 10.4. Después de que la reducción de intensidad se aplica a los lazos internos alrededorde B2 y B3, el único uso de i y j es determinar el resultado de la prueba en el bloque B4. Se sabe que los valores de i y t2 cumplen la relación t2 = 4 \* i, en tanto que los de j y t4 satisfacen larelación t4 = 4\*j, así que la prueba t2 >= t4 es equivalente a i >= j. Una vez hecha esta sustitución, i en el bloque B2 y j en el bloque B3 se convierten en variables inactivas y las asignaciones a ellas en estos bloques se convierten en código inactivo que puede ser eliminado, obteniéndose el grafo de flujo que se muestra en la figura 10. 10.

Las transformaciones para mejorar el código han sido efectivas. En la figura 10. 10, el número de instrucciones en los bloques B2 y B3 se ha reducido de 4 a 3 a partir del grafo de flujo original de la figura 10.5, en B5 se ha reducido de 9 a 3, y en B6 de 8 a 3. En realidad, B1 ha aumentado de cuatro instrucciones a seis, pero B1 se ejecuta sólo una vez en el fragmento, así que el tiempo de ejecución total está rara vez influido por el tamaño de B1.

**OPTIMACION DE BLOQUES BASICOS**

En el capítulo 9 se vieron varias transformaciones para mejorar el código para los bloques básicos. Estas incluyen transformaciones que preservan la estructura, como la eliminación de subexpresiones comunes y la eliminación de código inactivo, y las transformaciones algebraicas como la reducción de intensidad.

Muchas de las transformaciones que preservan la estructura se pueden implantar mediante laconstrucción de un GDA para un bloque básico. Recuérdese que hay un nodo en el GDA para cada uno de los valores iniciales de las variables que aparecen en el bloque básico, y hay un nodo n asociado a cada proposición s dentro del bloque. Los hijos de n son los nodos correspondientes a proposiciones que son las últimas definiciones anteriores a s de los operandos utilizados por s. El nodo n se etiqueta con el operador aplicado a s, y también se asocia a n la lista de variables para las cuales es la última definición dentro del bloque. También se tienen en cuenta los nodos,si los hay, cuyos valores están activos a la salida del bloque; estos son los nodos de salida.

Las subexpresiones comunes se pueden detectar observando, cuando un nuevo nodo m está apunto de añadirse, si existe un nodo n con los mismos hijos, en el mismo orden, y con el mismo operador. Si es así, n calcula el mismo valor que m y puede ser utilizado en su lugar,

Ejemplo En la figura 10. 11 se muestra un GDA para el bloque (10. 3)

a : = b + c

b : = a - d

c : = b + c

d : = a – d

Cuando se construye el nodo para la tercera proposición c: =b+c, se sabe que el uso de b en b+c se refiere al nodo de la figura 10. 11 etiquetado con -, porque esa es la definición más reciente de b. Por tanto, no se confunden los valores calculados en las proposiciones una y tres.

Sin embargo, el nodo correspondiente a la cuarta proposición d:= a - d tiene el operador y los nodos etiquetados con a y d0 como hijos. Como el operador y los hijos son los mismos que parael nodo correspondiente a la proposición dos, no se crea este nodo sino que se añade d a la lista de definiciones para el nodo etiquetado con

Puede suceder que, como sólo hay tres nodos en el GDA de la figura 10. 11, el bloque (10.3) se puede sustituir por un bloque con sólo tres proposiciones. De hecho, si b o d no está activa a la salida del bloque, entonces no hay que calcular esa variable, y se puede utilizar la otra para recibir el valor representado por el nodo etiquetado con - en la figura 10. 11. Por ejemplo, si b noestá activa a la salida, se podría utilizar:

a : = b + c

d : = a – d

c : = d + c

Sin embargo, si tanto b como d estan activas a la salida, entonces se debe utilizar una cuarta proposición para copiar el valor de una a la otra.

Obsérvese que cuando se buscan subexpresiones comunes, en realidad se están buscando expresiones que vayan a calcular elmismo valor, independientemente cómo se calcule dicho valor. Por tanto, el método del GDA no tendrá en cuenta el hecho de que la expresión calculada por la primera y la cuarta proposiciones en la secuencia.

a : = b + c

b : = b - d

c : = c + d

e : = b + c

es la misma, es decir, b+c. Sin embargo, las identidades algebraicas aplicadas al GDA. como seestudia a continuación, pueden exponer la equivalencia. En la figura 10. 1 se muestra el GDA para esta secuencia.

La operación sobre GDA que corresponde a la eliminación del código inactivo es bastante directa de implantar. Se borra de un GDA cualquier raíz (nodo sin ancestros) que no tenga variables activas. La aplicación repetida de esta transformación eliminará todos los nodos del GDA que correspondan a código inactivo.

**El uso de identidades algebraicas**

Las identidades algebraicas representan otra clase importante de optirnaciones sobre bloques básicos. En la sección 9.9 se introdujeron algunas sencillas transformaciones algebraicas que sepueden intentar durante la optimación. Por ejemplo, se pueden aplicar identidades aritméticas,como

x + 0 = 0 + x = x

x - 0 = x

x \* 1 = 1\* x = x

x / x = 1

Otra clase de optimaciones algebraicas incluyen la reducción de intensidad, es decir, sustituir unoperador más caro por otro más barato como en

x \*\* 2 = x \* x

2.0 \* x = x + x

x / 2 = x \* 0.5

Una tercera clase de optimaciones afines es el cálculo previo de constantes. Aquí se evalúan las expresiones constantes durante la compilación y se sustituyen las expresiones constantes por sus valores. Por tanto, la expresión 2 \* 3 . 14 se sustuiría por 6. 2 8. Muchas expresiones constantes surgen con el uso de constantes simbólícas.

El proceso de construcción de un GDA puede ayudar a aplicar estas y otras transformaciones algebraicas más generales, como la conmutatividad y la asociatividad. Por ejemplo, supóngaseque \* es conmutativo; es decir, x\*y = y\*x. Antes de crear un nuevo nodo etiquetado con \* con hijo izquierdo m e hijo derecho n, se revisa si ya existe dicho nodo. Después se busca un nodoque tenga operador \*, hijo izquierdo n e hijo derecho m.

Los operadores relacionales <=, >=, <, >, = y != a veces generan subexpresiones comunes imprevistas. Por ejemplo, la condición x>y también se puede comprobar restando los argumentos y realizando una prueba del código de condición establecido por la resta. (Sin embargo, la resta puede introducir desbordamientos y desbordamientos negativos, mientras que una instrucción de comparación no lo haría.) Por tanto, sólo hay que generar un nodo del GDA para x-y y x>y.

Las leyes asociativas también se pueden aplicar para evidenciar las subexpresiones comunes. Porejemplo, si el código fuente tiene las asignaciones

a := b + c

e := c + d + b

se podría generar el siguiente código intermedio:

a :=b + c

t := c + d

e := t + b

Si no se necesita t fuera de este bloque, se puede cambiar esta secuencia por

a := b + c

e := a + d

utilizando la asociatividad y la conmutatividad de +.

El escritor del compilador debe examinar cuidadosamente la especificación de lenguaje paradeterminar los ordenamientos de cálculos que están permitidos, puesto que la aritmética del computador no siempre obedece las identidades algebraicas de la matemática. Por ejemplo, el estándar para FORTRAN 77 establece que un compilador puede evaluar cualquier expresión matemáticamente equivalente si no se viola la integridad de los paréntesis. Por tanto, un compilador puede evaluar x\*y-x\*z como x\* (y-z) pero no puede evaluar a+ (b-c) como (a+b) -c. Un compilador de FORTRAN debe por tanto llevar un registro de dónde estaban los paréntesis en las expresiones del lenguaje fuente, si va a optimar programas según la definición del lenguaje.

**LAZOS EN LOS GRAFOS DE FLUJO**

Antes de considerar optimaciones de lazos, hay que definir lo que constituye un lazo en un grafo de flujo. Se utilizará la noción de que un nodo "dornina" a otro para definir los Iazos naturales" y la importante clase especial de grafos de flujo "reducibles". En la sección 10.9 se dará un algoritmo para encontrar dominantes y comprobar la reducibilidad de los grafos de flujo.

**Dominantes**

Se dice que el nodo d de un grafo de flujo domina al nodo n, que se escribe d dom n, si todo camino desde el nodo inicial del grafo de flujo a n pasa por d. Con esta definición, todo nodo se domina a sí mismo, y la entrada de un lazo (como se definió en la Sec. 9.4) domina todos los nodos del lazo.

Ejemplo 10.6. Considérese el grafo de flujo de la figura 10. 13, con nodo inicial 1. El nodo inicial domina a todos los nodos. El nodo 2 sólo se domina a sí mismo.

puesto que el control puede llegar a cualquier otro nodo a través de un camino que comience con 1 -> 3. El nodo 3 domina a todos excepto a 1 y a 2. El nodo 4 domina a todos excepto a 1, 2 y 3, porque todos los caminos a partir de 1 deben comenzarcon 1 ->2-> 3->4 o con 1 - > 3 -> 4. Los nodos 5 y 6 sólo se dominan a ellos mismos, porque el flujo del control puede saltar a cualquiera de ellos a través del otro. Por último, 7 domina a 7, 8, 9, 10; 8 domina a 8, 9, 10; 9 y 10 sólo se dominan a ellos mismos.

Una forma útil de presentar la información sobre dominadores es en un árbol, llamado árbol dedominación, en el que el nodo inicial es la raíz, y cada nodo d domina sólo a sus descendientes enel árbol. Por ejemplo, en la figura 10. 14 se muestra el árbol de dominación para el grafo de flujode la figura 10. 13.

La existencia de árboles de dominación se deriva de una propiedad de los dominantes; cada nodo n tiene un dominador inmediato m que es el último dominador de n en cualquier camino desde el nodo inicial a n. Respecto a la relación dom, el dominador inmediato m tiene la propiedad de que si d != n y d dom n, entonces d dom m.

**Lazos naturales**

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

1 . Un lazo debe tener un solo punto de entrada, llamado "encabezamiento". Este punto de entrada domina

todos los nodos dentro del lazo, o no sería la única entrada al lazo.

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

encabezamiento.

Una buena forma de encontrar todos los lazos en un grafo de flujo es buscar aristas en el grafo de flujo cuyas cabezas dominen sus colas. (Si a -> b es una arista, b es la cabeza y a es la cola.) Dichas aristas se denominan aristas de retroceso.

Ejemplo 10.7. En la figura 10. 13 hay una arista 7 -> 4, y 4 dom 7, De manera similar, 10 -> 7 es una arista, y 7 dom 10. Las otras aristas con esta propiedad son 4 -> 3, 8 ->3 y 9 -> 1. Obsérvese que éstas son exactamente las aristas que intuitivamente parecen formar lazos en el grafo de flujo.

Dada una arista de retroceso n -> d, se define el lazo natural de la arista como d más el conjunto de nodos que puede alcanzar n sin ir a través de d. El nodo d es el encabezamiento del lazo.

Ejemplo 10.8. El lazo natural de la arista 10 -> 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 lazo natural de 9 - > 1 es el grafo de flujo completo. (No hay que olvidar el camino 10 -> 7 -> 8 -> 9.)

**Algorítmo** Construcción del lazo natural de una arista de retroceso.

*Entrada*. Un grafo de flujo G y una arista de retroceso n -> d.

*Salida*. El conjunto lazo que consta de todos los nodos dentro del lazo natural de n -> d.

*Método*. Comenzando con el nodo n, se considera cada nodo m != d que se sabe que está en lazo para asegurarse de que los predecesores de m también se colocan en lazo. El algoritmo se da en lafigura 10. 15. Cada nodo en lazo, excepto d, se coloca una vez en pila, así que sus predecesores serán examinados. Obsérvese que como d se coloca en el lazo al inicio, nunca se examinan sus precedesores, y por tanto se encuentran sólo aquellos nodos que alcanzan n sin pasar por d.

**Lazos internos**

Si se utilizan los lazos naturales como “los lazos", entonces existe la propiedad útil de que, a menos que dos lazos tengan el mismo encabezamiento, son disjuntos o

**procedure** inserta (m);

**if** m no está en lazo **then begin**

lazo:= lazo u {m};

mete m en pila

**end**;

/\* sigue el programa principal \*/

pila := vacía;

lazo := {d};

inserta (n);

**while** pila no sea vacía **do begin**

saca m, el primer elemento de pila, de pila;

**for** cada predecesor b de m **do** inserta (p)

**end**

uno está completamente contenido (anidado dentro) en el otro. Por tanto, sin tener en cuenta por el momento los lazos con el mismo encabezamiento, se tiene una noción natural de lazo interno:uno que no contiene otros lazos.

Cuando dos lazos tienen el mismo encabezamiento, como en la figura 10. 16, es difícil saber cuál es el lazo interno. Por ejemplo, si la condicional al final de B1 fuera

if a = 10 goto B2

probablemente el lazo {B0, B1, B3} sería el lazo interno. Sin embargo, no es posible estar segurosin un examen detallado del código. Quizás a casi siempre es 10, y es habitual iterar el lazo {B0, B1, B2} muchas veces antes de ir a B3. Por tanto, se supondrá que cuando dos lazos naturales tienen el mismo encabezamiento pero ninguno está anidado dentro del otro, se combinan y consideran como un solo lazo.

**Preencabezamientos**

Varias transformaciones exigen trasladar las proposiciones "antes del encabezamiento". Por tanto,se comienza el tratamiento de un lazo L creando un nuevo bloque, llamado preencabezamiento. El preencabezamiento sólo tiene el encabezamiento corno sucesor y todas las aristas que antes entraban al encabezamiento de L desde fuera de L ahora entran al preencabezamiento. No se modifican las aristas desde dentro del lazo L al encabezamiento. El ordenamiento se muestra en lafigura 10. 17. Al inicio, el preencabezamíento está vacío pero las transformaciones sobre L pueden colocar proposiciones en él.

**Grafos de flujo reducibles**

Los grafos de flujo que ocurren en la práctica, a menudo forman parte de la clase de grafos de flujo reducibles que se definen más adelante. El uso exclusivo de proposiciones de flujo de control estructuradas como las proposiciones **if-then-else**, **whiledo**, **continue** y **break** produce programas cuyos grafos de flujo siempre son reducibles. Incluso los programas escritos utilizando proposiciones goto por programadcres sin conocimiento previo de diseño de programas estructurados son reducibles casi siempre.

Se han propuesto muchas definiciones de "grafo de flujo reducible". La que aqui se adopta tiene una de las propiedades más importantes de los grafos de flujo reducibles: que no hay saltos hacia dentro de los lazos desde el exterior; la única entrada a un lazo es a través de su encabezamiento.

Un grafo de flujo G es reducible si, y sólo si, se pueden particionar las aristas en dos grupos disjuntos, llamadas con frecuencia aristas de avance y aristas de retroceso, con las siguientes dos propiedades:

1. Las aristas de avance forman un grafo acíclico en el que cada nodo se puede alcanzar desde el nodo inicial de G.

2. Las aristas de retroceso constan sólo de las aristas cuyas cabezas dominan sus colas.

Ejemplo 10.9. El grafo de flujo de la figura 10. 13 es reducible. En general, si se conoce la relación dom para un grafo de flujo, se pueden encontrar y eliminar todas las aristas de retroceso. Las aristas restantes deben ser las aristas de avance si el grafo es reducible, y para comprobar si un grafo de flujo es reducible, basta con comprobar si las aristas de avance forman un grafo acíclico. En el caso de la figura 10. 13. es fácil comprobar que si se eliminan las cinco aristas deretroceso 4 -> 7, 7 -> 4, 8 -> 3, 9 -> 1 y 10 -> 7, cuyas cabezas dominan a sus colas, el grafo restante es acíclico. 1

Ejemplo 10.10. Considérese el grafo de flujo de la figura 10. 18, cuyo nodo inicial es 1. Este grafo de flujo no tiene aristas de retroceso, puesto que ninguna cabeza de una

arista domina a la cola de dicha arista. Por tanto, sólo podría ser reducible si el grafo completo fuera acíclico. Pero como no lo es, el grafo de flujo no es reducible. De manera intuitiva, la razón de que este grafo de flujo no sea reducible es que se puede entrar al lazo 2-3 por dos sitios diferentes, los nodos 2 y 3.

La propiedad clave de los grafos de flujo reducibles para el análisis de lazos es que en dichos grafos de flujo cada conjunto de nodos que informalmente podría ser considerado como un lazo debe contener una arista de retroceso. De hecho, basta con examinar los lazos naturales de aristas de retroceso para encontrar todos los lazos en un programa cuyo grafo de flujo sea reducible. Porel contrario, el grafo de flujo de la figura 10. 18 tiene un "lazo" que consta de los nodos 2 y 3,pero no hay ninguna arista de retroceso de la cual sea éste su lazo natural. De hecho, ese "lazo"tiene dos encabezamientos, 2 y 3, lo que hace que la aplicación de muchas técnicas de optimación de código, como las presentadas en la sección 10.2 para traslado de código y eliminación de variables de inducción, no sean aplicables directamente.

Por fortuna, las estructuras de flujo de control no reducibles, como la de la figura 10. 18, aparecen tan raramente en la mayoría de los programas que convierten el estudio de lazos con más de un encabezamiento en poco importante. Incluso hay lenguajes, como BLISS y MODULA 2, que permiten sólo programas con grafos de flujo reducibles, y muchos otros lenguajes producirán sólo grafos de flujo reducibles mientras no se utilicen proposiciones **goto**.

Ejemplo 10.11. , Volviendo a la figura 10. 13, se observa que el único "lazo interno", es decir, un lazo sin sublazos, es {7, 8, l0}, el lazo natural con arista de retroceso 10 -> 7. El conjunto {4, 5,6, 7, 8, 10} es el lazo natural de 7 -> 4. (Obsérvese que 8 y 10 pueden alcanzar 7 a través de la arista 10 -> 7.) La intuición de que {4, 5, 6, 7} forma un lazo es equivocada, ya que 4 y 7 serían entradas desde el exterior, incumpliendo el requisito de una sola entrada. Dicho de otro modo, no hay razón para suponer que el control emplea mucho tiempo en ir alrededor del conjunto denodos {4, 5, 6, 7}; es tan plausible que el control pase a 8 desde 7 más veces que a 4. Incluyendo 8 y 10 en el lazo, se tiene mayor certeza de haber aislado una región muy transitada del programa.

Sin embargo, hay que reconocer el peligro de hacer suposiciones sobre la frecuencia de las ramificaciones. Por ejemplo, si se saca una proposición invariante de 8 o 10 en el lazo {7, 8, 1 0}y de hecho, el control siguiera la arista 7 -> 4 en más ocasiones que 7 -> 8, se incrementaría realmente el número de veces que se ejecutara la proposición trasladada. En la sección 10.7 se estudiarán métodos para evitar este problema.

Después, el lazo más grande es {3, 4, 5, 6, 7, 8, l0}, que es el lazo natural de ambas aristas 4 -> 3 y 8 -> 3. Como antes, la intuición de que {3, 4} debería considerarse como un lazo incumple el requisito de un solo encabezamiento. El último lazo, el de la arista de retroceso 9 ->1, es el grafo de flujo completo.

Hay varias propiedades útiles adicionales de los grafos de flujo reducibles, que se presentarán cuando se estudien los temas de búsqueda en profundidad y análisis de intervalos en la sección 10.9.

**INTRODUCCION AL ANALISIS GLOBAL DEL FLUJO DE DATOS**

Para realizar la optimación de código y un buen trabajo de generación de código. un compilador necesita reunir información sobre el programa como un todo y distribuir esta información a cada bloque en el grafo de flujo. Por ejemplo, en la sección 9.7 se vio que el conocimiento de las variables que están activas a la salida de cada bloque puede optimar el uso de los registros. En la sección 10.2 se sugirió cómo utilizar el conocimiento de las subexpresiones comunes para eliminar cálculos redundantes. Asimismo, en las secciones 9.9 y 10.3 se estudió cómo un compilador puede aprovechar las "definiciones de alcance", como saber dónde se definió por última vez una variable como depura antes de llegar a un bloque dado, para poder realizar transformaciones como el cálculo previo de constantes y la eliminación de código inactivo. Estos hechos son sólo unos cuantos ejemplos de la información de flujo de datos que un compilador optimador recopila mediante un proceso llamado análisis de flujo de datos.

La información del flujo de datos se puede recopilar estableciendo y resolviendo sistemas de ecuaciones que relacionan la información en varios puntos de un programa. Una ecuación típica tiene la forma

sal [S] = gen [S] u (ent [S] - desact [S])

y se puede leer como Ia información al final de una proposición se genera dentro de la proposición o se introduce al comienzo y no se desactiva cuando el control fluye por la proposición". Dichas ecuaciones se denominan ecuaciones deflujo de datos.

Los detalles sobre cómo se plantean y resuelven las ecuaciones de flujo de datos dependen de tres factores:

1 . Las nociones de generar y desactivar dependen de la información deseada, es decir, del problema de análisis del flujo de datos que debe resolverse. Además. para algunos problemas, en lugar de avanzar junto con el flujo del control y definir sal [S] según ent [S], hay que hacerlo hacia atrás y definir ent [S] según sal [S].

2. Como los datos fluyen a lo largo de caminos de control, el análisis del flujo de datos está influido por las construcciones de control en un programa. De hecho. cuando se escribe sal[S] se asume implícitamente que hay un punto final único donde el control sale de la proposición; engeneral, las ecuaciones se establecen al nivel de bloques básicos en lugar de proposiciones,porque los bloques tienen puntos finales únicos.

3. Algunas sutilezas acompañan dichas proposiciones como las llamadas a procedimientos, las asignaciones por medio de variables tipo apuntador e incluso las asignaciones a variables detipo matriz.En esta sección se considera el problema de determinar el conjunto de definiciones que alcanzan un punto en un programa y su uso para encontrar oportunidades de hacer cálculo previo de constantes. Más adelante en este capítulo, los algoritmos para traslado de código y la eliminaciónde variables de inducción utilizarán también esta información.

Inicialmente se consideran programas construidos utilizando proposiciones **if** y **do-while**. El flujo de control previsible en estas proposiciones permite concentrarse en las ideas necesarias para establecer y resolver las ecuaciones de flujo de datos. Las asignaciones en esta sección son proposiciones de copia o de la forma a: =b+c. En este capítulo con frecuencia se utiliza "+" como operador típico. Todo lo que se indique sirve directamente para otros operadores, incluidos aquellos con un operando o con más de dos operandos.

**Puntos y caminos**

Dentro de un bloque básico, se habla del punto entre dos proposiciones adyacentes, al igual que del punto antes de la primera proposición y después de la última. Por tanto, el bloque B1 de lafigura 10. 19 tiene cuatro puntos: uno antes de todas las asignaciones y uno después de cada una de las tres asignaciones.

A continuación se considera una visión global y todos los puntos en la totalidad de los bloques. Un camino desde p1 a pn, es una secuencia de puntos p1, p2, ..., pn tal que para cada i entre 1 y n – 1,

1 . pi es el punto que precede inmediatamente a una proposición y pi+ 1 es el punto que sigue inmediatamente a dicha proposición en el mismo bloque, o

2. pi es el final de un bloque y pi+ 1 es el comienzo de un bloque sucesor.

Ejemplo 10.12. En la figura 10. 19 hay un camino desde el comienzo del bloque B5 hasta elcomienzo del bloque B6. Pasa por el punto final de B5 y después por todoslos puntos en B2, B3 y B4, en orden, antes de alcanzar el comienzo de B6.

**Definiciones de alcance**

Una definición de una variable x es una proposición que asigna, o puede asignar, un valor a x. Las formas más comunes de definición son las asignaciones a x y las proposiciones que leen unvalor de un dispositivo de E/S y lo almacenan en x. Estas proposiciones ciertamente definen unvalor para x, y se consideran definiciones no ambiguas de x. Otras clases de proposiciones pueden definir un valor para x; se denominan definiciones ambiguas. Las formas más habituales de definiciones ambiguas de x son:

1 . Una llamada a un procedimiento con x como parámetro (que no sea un parametro por valor) o un procedimiento que puede acceder a x porque x está dentro del ámbito del procedimiento. También hay que considerar la posibilidad de los "sinónimos", donde x no está dentro del ámbito del procedimiento, pero x se ha identificado con otra variable que se pasa como parámetro o que está dentro del ámbito. Estos aspectos se estudian en la sección 10.8.

2. Una asignación por medio de un apuntador que pudiera referirse a x. Por ejemplo, laasignación \*q: =y es una definición de x si es posible que q apunte a x. En la sección 10.8 también se estudian los métodos para determinar hacia dónde podría apuntar un apuntador, pero en ausencia de un conocimiento en sentido contrario, se debe asumir que una asignación por medio de un apuntador es una definición de todas las variables.

Se dice que una definición d alcanza un punto p si hay un camino desde el punto que sigue inmediatamente d hasta p, tal que d no se "desactive" a lo largo del camino. Intuitivamente, si una definición d de una variable a alcanza el punto p, entonces d debe ser el lugar en el que el valor de a utilizado en p puede haber sido definido por última vez. Se desactiva una definición de una variable a si entre dos puntos a lo largo del camino hay una definición de a. Obsérvese que sólo las definiciones no ambiguas de a desactivan otras definiciones de a. Por tanto, un punto puede ser alcanzado por una definición no ambigua y por una definición ambigua de la misma variable que aparezca posteriormente a lo largo de un camino.

Por ejemplo, ambas definiciones, i:=m-1 y j:=n, en el bloque B1 de la figura 10. 19 alcanzan el principio del bloque B2, al igual que la definición j : =j-1 suponiendo que no haya asignaciones o lecturas de j en B4, B5, o en la parte de B3 que sigue a esta definición. Sin embargo, la asignación j en B3 desactiva la definición j : =n, así que esta última no alcanza B4, B5 o B6

Si se formulan así las definiciones de alcance, a veces surgen imprecisiones. Sin embargo, todas van en dirección "segura" o "conservadora". Por ejemplo, obsérvese nuestra suposición de que sepueden recorrer todas las aristas de un grafo de flujo. Esto puede no ser cierto en la práctica. Por ejemplo, el control puede realmente alcanzar la asignación a: = 4 en el siguiente fragmento de programa para ningún valor de a y de b.

if a = b then a := 2

else if a = b then a := 4

Decidir en general si se puede tomar cada camino en un grato de flujo es un problema indecidible, y no se intentará resolverlo.

Un tema recurrente en el diseño de las transformaciones para mejorar el código es que, ante la duda, Sólo se deben tomar decisiones conservadoras, aunque las estrategias conservadoras pueden impedir la realización de transformaciones que no suponen ningún peligro. Una decisión es conservadora si nunca conlleva un cambio en lo que calcula el programa. En aplicaciones de definiciones de alcance, es conservador asumir que una definición puede alcanzar un punto, aunque no lo haga. Por tanto, se permiten caminos que pueden no ser nunca recorridos durante la ejecución de un programa y se permite que las definiciones atraviesen definiciones ambiguas de la misma variable.

**Análisis de flujo de datos de programas estructurados**

Los grafos de flujo para las construcciones de flujo del control como las proposiciones **do-while** tienen una propiedad útil: hay un solo punto de comienzo en el que el control entra y un solo punto final por el que sale el control cuando termina la ejecución de la proposición. Esta propiedad se explota cuando se habla de que las definiciones alcanzan el comienzo y el final de proposiciones con la siguiente sintaxis:

S → **id** : = E | S ; S | **if** E **then** S **else** S | **do** S **while** E

E → **id** + **id** | **id**

Las expresiones de este lenguaje son similares a las del código intermedio, pero los grafos deflujo para las proposiciones tienen formas limitadas sugeridas por los diagramas de la figura10.20. Un propósito principal de esta sección es estudiar las ecuaciones de flujo de datos resumidas en la figura 10.2 1.

Se define una parte de un grafo de flujo denominada región como un conjunto de nodos N que incluye un encabezamiento, que domina a todos los otros nodos de la región. Todas las aristas entre los nodos de N están en la región, excepto (tal vez) alguno que entra al encabezamiento .La parte de un grafo de flujo correspondiente a una proposición S es una región que cumple la limitación adicional de que el control puede fluir sólo a un bloque exterior cuando abandona laregión.

Por conveniencia técnica se supone que hay bloques falsos sin proposiciones (indicados porcírculos abiertos en la Fig. 10.20) a través de los cuales el control fluye justo antes de entrar y justo antes de abandonar la región. Se dice que los puntos de comienzo de los bloques ficticios a la entrada y la salida de la región de una proposición son los puntos comienzo y fin, respectivamente, de la proposición.

Las ecuaciones de la figura 10.21 son una definición inductiva, o dirigida por la sintaxis, de losconjuntos ent [S], sal [S] y desact [S] para todas las proposiciones S. Los conjuntos gen [S] y desact [S] son atributos sintetizados; se calculan en forma ascendente, desde las proposiciones más pequeñas a las más grandes. Se desea que

la definición d esté en gen [S] si d alcanza el fin de S, independientemente de que alcance o no el comienzo de S. Dicho de otra forma, d debe aparecer en S y alcanzar el fin de S a través de un camino que no salga de S. Esta es la justificación para señalar que gen [S] es el conjunto dedefiniciones "generadas por S".

De manera similar, se pretende que desact [S] sea el conjunto de definiciones que nunca alcanzan el fin de S, aunque alcancen el comienzo. Por tanto, tiene sentido considerar estas definiciones como "desactivadas por S". Para que la definición d esté en desact [S], cada camino desde elcomienzo al fin de S debe tener una desición no ambigua de la misma variable definida por d, y si d aparece en S, entonces debe haber otra definición de la misma variable siguiendo cada ocurrencia de d a lo largo de cualquier camino.

Las reglas para gen y desact, siendo traducciones sintetizadas, son relativamente faciles de entender. Para comenzar, obsérvense las reglas de la figura 10. 2 1 (a) para una asignación simple a la variable a. Evidentemente, esta asignación es una definición de a, por ejemplo, la definición d. Entonces d es la única definición que alcanza el fin de la proposición independientemente deque alcance el comienzo. Por tanto,

gen [S] = {d}

Fór otra parte, d "desactiva" todas las otras definiciones de a, así que se escribe

desact [S] = Da {d}

donde Da es el conjunto de todas las definiciones en el programa para la variable a.

La regla para una cascada de proposiciones, que se ilustra en la figura 10. 2 1 (b), es un poco más sutil. ¿En qué circunstancias la definición d es generada por S = S1 ; S2 ? Primero, si es generada por S2, entonces es generada por S. Si d es generada por S1 alcanzará el fin de S suponiendo que S2 no la desactive. Así, se escribe

gen [S] = gen [S2] u (gen [S1] - desact [S2])

Se aplican razonamientos similares a la desactivación de una definición, de modo que se tiene

desact [S] = desact[S2] u (desact [S1] -gen [S2])

Para la proposición **if**, ilustrada en la figura 10.2 1 (c), se observa que si cualquiera de las ramas del "if”' genera una definición, entonces esa definición alcanza el fin de la proposición S. Por tanto,

gen [S] = gen [S1] u gen [S2]

Sin embargo, para "desactivar" una definición d, la variable definida por d se debe desactivar a lo largo de cualquier camino que vaya desde el comienzo al final de S. Además, se debe desactivar alo largo de una de las ramas, de modo que

desact [S] = desact [S1] n desact [S2]

Por último, considérense las reglas para los lazos de la figura 10. 2 1 (d). Puesto de manera sencilla, el lazo no afecta a gen o desact. Si la definición d se genera dentro de S1, entonces alcanza el final de S1 y el final de S. A la inversa, si d se genera dentro de S, sólo puede ser generada dentro de S1. Si S1 desactiva d, entonces recorrer el lazo no servirá de nada; se redefine la variable de d dentro de S1 cada vez que se hace un recorrido. A la inversa, si S desactiva d, entonces debe ser desactivada por S1. La conclusión es:

gen [S] = gen [S1]

desact [S] = desact [S1]

**Estimación conservadora de la información sobre el flujo de datos**

Hay un error de cálculo sutil en las reglas para gen y desact de la figura 10.2 1. Se ha dado por supuesto que la expresión condicional E en las proposiciones **if** y **do** no se interpretan; es decir, existen entradas al programa que hacen que sus ramas vayan a una de las dos partes. Dicho de otra forma, se supone que cualquier camino de la teoría de grafos en el diagrama de flujo es también un camino de ejecución, es decir. un camino que se recorre cuando se ejecuta el programa con al menos una entrada posible.

Este no siempre es el caso, y de hecho no se puede decidir en general si se puede o no tomar una ramificación. Supóngase, por ejemplo, que la expresión E en una proposición **if** fuera siempre verdadera. Entonces nunca podría tomarse el camino a través de S2 en la figura 10.2 1 (c). Esto tiene dos consecuencias. Primero, una definición generada por S2, no realmente generada por S,porque no hay forma de llegar desde el comienzo de S a la proposición S2. Segundo, ninguna definición dentro de desact [S] puede alcanzar el fin de S. Por tanto, cada definición de este tipo deberia estar lógicamente en desact [S], aunque no esté en desact [S2].

Cuando se compara el conjunto gen calculado con el "verdadero" gen, se descubre que el gen verdadero es siempre un subconjunto del gen calculado. Por otra parte, el desact verdadero es siempre un supraconjunto del desact calculado. Estas contenciones se aplican incluso después deconsiderar las otras reglas de la figura 10.2 1. Por ejemplo, si la expresión E en una proposición **do**-S-**while**-E nunca puede ser falsa, entonces nunca se saldrá del lazo. Por tanto, el gen verdadero es 0, y toda definición es desactivada por el lazo. El caso de una cascada de proposiciones, en la figura 10.21(b), donde se debe tener en cuenta la imposibilidad de salir de S1 o de S2, debido a un lazo infinito, se deja como ejercicio práctico.

Es natural preguntarse si estas diferencias entre los conjuntos gen y desact verdaderos y calculados obstaculizan el análisis del flujo de datos. La respuesta está en el uso que se haga de esos datos. En el caso de las definiciones de alcance, normalmente se utiliza la información para inferir que el valor de una variable x en un punto se limita a un número pequeño de posibilidades. Por ejemplo, si resulta que las únicas definiciones de x que alcanzan este punto son de la forma x: = 1, se puede inferir que x tiene el valor 1 en ese punto. Por tanto, se puede decidir sustituir lasreferencias a x por referencias a 1.

Como consecuencia, no parece serio sobreestimar el conjunto de definiciones que alcanzan unpunto; simplemente impide realizar una optimación que se podría hacer legítimamente. Por otraparte, subestimar el conjunto de definiciones es un error fatal; podría conducir a un cambio en el programa que modifique lo que el programa calcula. Por ejemplo, se puede pensar que todas las definiciones de alcance de x dan a x elvalor de 1, y por tanto se sustituye x por 1; pero existe otra definición de enlace no detectada queda a x el valor de 2. Entonces, para el caso de las defies de alcance, se dice que un conjunto dedefiniciones es seguro o conservador estimación es un supraconjunto (no necesariamente unsupraconjunto propio) conjunto verdadero de definiciones de alcance. La estimación se denomina insi no esnecesariamente un supraconjunto del verdadero.

Para cada problema de flujo de datos se debe examinar el efecto de las estimaciones imprecisas sobre los tipos de cambios que puedan causar en los programas. Generalmente se aceptan discrepancias que sean seguras en el sentido de que pue prohibir optimaciones que se podrían hacer legalmente, pero nose aceptan discrepancias que sean inseguras en el sentido de que puedan causar "optimaciones" que no preserven el comportamiento observado desde el exterior del programa. En cada problema de flujo dedatos, normalmente un conjunto o un supraconjunto (pero no ambos) de la respuesta verdadera es seguro.

Volviendo a las implicaciones de seguridad sobre la estimación de gen y desact para las definiciones de alcance, obsérvese que las discrepancias, supraconjuntos para gen y subconjuntos para desact están ambas en la dirección segura. Intuitivamente, incrementar gen aumenta el conjunto de definiciones que pueden alcanzar un punto, y no puede impedir que una definición alcance un lugar que realmente alcanzó, Asi mísmo, reducir desact sólo puede incrementar el conjunto de definiciones que alcanzan un punto dado.

**Calculo de ent y sal**

Muchos problemas de flujo de datos se pueden resolver mediante traducciones sinsetizadas similares a las utilizadas para calcular gen y desact. Por ejemplo, se puede pretender determinar,para cada proposición S, el conjunto de variables definidas dentro de S. Esta información sepuede calcular con ecuaciones análogas a las de gen, sin ni siquiera ser necesarios conjuntos análogos a desact. Se puede utilizar, por ejemplo, para determinar cálculos de lazo invariante.

Sin embargo, hay otros tipos de información sobre el flujo de datos, como el problema de las definiciones de alcance que fue utilizado como ejemplo, donde también hay que calcular algunos atributos heredados. Resulta que ent es un atributo heredado y que sal es un atributo sintetizado que depende de ent. Se pretende que ent [S] sea el conjunto de definiciones que alcanzan elcomienzo de S teniendo en cuenta el flujo del control a lo largo de todo el programa, incluidas las proposiciones fuera de S o dentro de la que S está anidada. El conjunto sal [S] se define de manera similar para el final de S. Es importante observar la distinción entre sal[S] y gen [S]. Este último es el conjunto de definiciones que alcanzan el final de S sin seguir carninos fuera de S.

Como ejemplo de la diferencia, considérese la cascada de proposiciones en la figura 10.21 (b). Una proposición d puede ser generada en S1 y por tanto alcanzar el comienzo de S2, Si d no está en desact [S2], d alcanzará el final de S2, y Por tanto estará en sal [S2]. Sin embargo, d no está en gen [S2].

Después de calcular gen [S] y desact [S] de forma ascendente, para todas las proposiciones S sepuede calcular ent y sal comenzando por la proposición que representa el programa completo,sabiendo que ent [S0l = 0 si S0 es el programa completo. Es decir, ninguna definición alcanza el principio del programa. Para cada uno de los cuatro tipos de proposiciones de la figura 10.2 1, se puede suponer que se conoce ent [S]. Se debe utilizar para calcular ent para cada una de las subproposiciones de S [lo cual es trivial en los casos (b) a (d) e irrelevante en el caso (a)].Después., recursivamente (de forma descendente) se calcula sal para cada una de las subproposiciones S1 o S2, y se utilizan estos conjuntos para calcular sal [S].

El caso más sencillo es la figura 10. 2 1 (a), donde la proposición es una asignación. Suponiendo que se conoce ent [S], se calcula sal mediante la ecuación (10.5). es decir

sal [S] = gen [S] u (ent [S] - desact [S])

En palabras, una definición alcanza el final de S si es generada por S (es decir, es la definición d que es la proposición), o alcanza el comienzo de la proposición y no es desactivada por la proposición.

Supóngase que se ha calculado ent [S] y que S es la cascada de dos proposiciones S1; S2, como en el segundo caso de la figura 10.21. Se comienza por observar que ent [S1] = ent [S2]. Después, recursivarnente se calcula sal [S1], que da ent [S2].puesto que una definición alcanza el comienzo de S2 si, y sólo si, alcanza el final de S1. Ahora se puede calcular recursivamente sal [S2], y este conjunto es igual a sal [S].

A continuación considérese la proposición if de la figura 10.2 1 (c). Como se asumió conservadoramente que el control puede fluir por cualquier rama, una definición alcanza el comienzo de S1 o S2 exactamente cuando alcanza el comienzo de S. Es decir,

ent [S1] = ent [S2] = ent [S]

También se deriva del diagrama de la figura 10. 2 1 (c) que una definición alcanza el final de S si,y sólo si, alcanza el final de una o de ambas subproposiciones; es decir.

sal [S] = sal [S1] u sal [S2]

Por tanto, se pueden utilizar estas ecuaciones para calcular ent [S1] y ent [S2] a partir de ent [S], calcular recursivamente sal [S1] y sal [S2], y después utilizar éstos para calcular sal [S].

**Utilización de lazos**

El último caso, la figura 10.21(d), presenta problemas específicos. Supóngase de nuevo que se tienen gen [S1] y desact [S1], habiéndolos calculado de forma ascendente, y supóngase que se tiene ent [S1] y que se está en el proceso de realizar un recorrido en profundidad del árbol de análisis sintáctico. A diferencia de los casos (b) y (c), no se puede utilizar simplemente ent [S] como ent [S1], porque las definiciones dentro de S1 que alcanzan el final de S1 pueden seguir el arco de regreso al comienzo de S1, y por tanto, estas definiciones también están en ent [S1]. En lugar de eso, se tiene

ent [S1] = ent [S] u sal [S1]

También se tiene la ecuación obvia para sal [S]:

sal[S] = sal[S1]

que se puede utilizar una vez que se haya calculado sal [S1]. Sin embargo, parece ue no se puede calcular ent [S1] por medio de (10.6) hasta que se haya calculado sal [S1], y el plan general ha sido calcular sal para una proposición, calculando primero ent para dicha proposición.

Por fortuna, hay una manera directa de expresar sal en términos de ent; viene dada por (10.5), o en este caso concreto:

sal [S1] = gen [S1] u (ent [S1] - desact [S1])

Es importante comprender lo que está ocurriendo en este caso. En realidad no se sabe que (10.7) es verdadera con respecto a una proposición arbitraria S1; sólo se sospecha que debe ser verdadera porque "tiene sentido" que una definición alcance el final de una proposición si, y sólo si, es generada dentro de la proposición o alcanza el comienzo y no es desactivada. Sin embargo,la única forma que se conoce de calcular sal para una proposición es mediante las ecuaciones dadas en la figura 10.21(a) a la (c). Se asume (10.7) y se derivan las ecuaciones para ent y sal dela figura 10.21(d). Después se pueden utilizar las ecuaciones de la figura 10.21(a) a la (d) para demostrar que (10.7) se cumple para una S1 arbitraria. Después se podrían reunir estas pruebas para hacer una demostración válida por inducción sobre el tamaño de una proposición S que la ecuación (10.7) y todas las ecuaciones de la figura 10.21 se cumplen para S y todas sus subproposiciones. No se hará así; se dejan las demostraciones como ejercicio, pero el razonamiento que se ha adoptado aquí debe resultar instructivo.

Incluso asumiendo (10.6) y (10.7) no se han resuelto todos los problemas. Estas dos ecuaciones definen una recurrencia para ent [S1] y sal [S1] simultáneamente. Las ecuaciones se reescribirán como

E = F u A

A = G u (E – D)

donde E, A, F, G y D corresponden a ent [S1], sal [S1] , ent [S], gen [S1] y desact [S1],respectivamente. Las dos primeras son variables, las otras tres son constantes.

Para resolver (10.8), supóngase que A = 0. Entonces se podría utilizar la primera ecuación de (10.8) para calcular una estimación de E, es decir,

E1 = F

A continuación se puede utilizar la segunda ecuación para obtener una mejor estimación de A:

A1 = G u (E1 - D) = G u (F – D)

Aplicando la primera ecuación a esta nueva estimación de A se obtiene:

E 2 = F u A1 = F u G u (F - D) = F u G

Si se reaplica la segunda ecuación, la siguiente estimación de A es:

A 2 = Gu(E2 - D) = G u (F u G - D) = G u (F - D)

Obsérvese que A2 = A1. Por tanto, sí se calcula la siguiente estimación de E. ser, igual a E1, lo cual dará otra estimación de A igual a A1, y así sucesivamente. De ese modo, los valores limitadores para E y A son los dados anteriormente por El y A1 .

Por tanto, se han obtenido las ecuaciones de la figura 10.2 1 (d), que son

ent [S1] = ent [S] u gen [S1]

sal[S] = sal[S1]

La primera de estas ecuaciones proviene del cálculo anterior; la segunda proviene del examen del grafo de la figura 10.2 1 (d).

Queda el detalle de por qué se pudo empezar por la estimación de A = 0. Recuérdese que en el estudio de las estimaciones conservadoras se sugirió que los conjuntos como sal [S1], al que representa A, deberían ser sobreestimados en lugar de subestimados. De hecho, si hubiera que comenzar con A = {d}, donde d es una definición que no aparece ni en F, G o D, entonces d concluiría en los valores limitadores de E y de A.

En este caso, se deben invocar los significados para ent y sal. Si tal d realmente perteneciera a ent[S1], tendría que haber un camino desde donde estuviera d al comienzo de S1 que mostrara cómo alcanza d ese punto. Si d estuviera fuera de S. entonces D tendría que estar en ent [S], en tanto que si d estuviera dentro de S (y por tanto dentro de S1) tendría que estar en gen [S1]. En el primer caso, d estaría en F y por tanto se colocaría en E mediante (10.8). En el segundo caso, d estaría en G y de nuevo se transmitiría a E a través de A en (10.8). La conclusión es que comenzar con una estimación demasiado pequeña y construir de forma ascendente añadiendo más definiciones a E yA es una forma segura de estimar ent [S1].

**Representación de conjuntos**

Los conjuntos de definiciones, como gen [S] y desact [S], se pueden representar de forma compacta utilizando vectores de bits. Se asigna un número a cada definición de interés en el grafóde flujo. Entonces el vector de bits que representa un conjunto de definiciones tendrá un 1 en la posición i si, y sólo si, la definición numerada con i está en el conjunto.

El número de una proposición de definición se puede tomar como el índice de la proposición en una matriz que contenga apuntadores a proposiciones. Sin embargo, no todas las definiciones pueden ser de interés durante el análisis global de flujo de datos. Por ejemplo, no hace falta asignar números a las definiciones de temporales utilizadas únicamente dentro de un solo bloque,como la mayoría de temporales generados para la evaluación de expresiones. Por tanto, los números de las definiciones de interés se registrarán normalmente en una tabla aparte.

Una representación de conjuntos mediante un vector de bits permitirá también que las operaciones de conjuntos se implanten efícientemente. La unión y la intersección de dos conjuntos se puede implantar mediante las operaciones lógicas **or** y **and**, respectivamente, que son operaciones básicas en la mayoría de los lenguajes de programación orientados a sistemas. La diferencia A - B de los conjuntos A y B se puede implantar tomando el complemento de B y utilizando después **and** lógico para calcular A /\ -B.

Ejemplo 10.13. En la figura 10.22 se muestra un programa con siete definiciones, indicadas por d1 hasta d7 en los comentarios a la izquierda de las definiciones. Los vectores de bits que representan los conjuntos gen y desact para las proposiciones de la figura 10.22 se muestran a la izquierda de los nodos del árbol sintáctico de la figura 10.23. Los mismos conjuntos fueron calculados aplicando las ecuaciones para flujo de datos de la figura 10.21 a las proposiciones representadas por los nodos del árbol sintáctico.

Considérese el nodo para d7 en el ángulo inferior derecho de la figura 10.23. El conjunto gen{d7} viene representado por 000 0001 y el conjunto desact {d1, d4} por 100 1000. Es decir, d7 desactiva todas las otras definiciones de i, su variable.

El segundo y tercer hijo del nodo **if** representan las partes **then** y **else**,respectivamente, del condicional.Observese que el conjunto gen000 0011en el nodo **if** es

la unión de los conjuntos 000 0010 y 000 0001 en el segundo y tercer hijos. El conjunto desact está vacío porque las definiciones desactivadas por las partes then y else son disjuntas.

Las ecuaciones de flujo de datos para una cascada de proposiciones se aplican al padre del nodo **if**. El conjunto desact en este nodo se obtiene por

000 0000 u (110 000 1 - 000 00 11) = 110 0000

En palabras, nada es desactivado por el condicional y d7, desactivado por la proposición d4, es generado por el condicional, de modo que sólo d1 y d2 están en el conjunto desact del padre del nodo if.

Ahora se puede calcular ent y sal, comenzando desde arriba en el árbol de análisis sintáctico. Se supone que el conjunto ent en la raíz del árbol sintáctico está vacío. Por tanto, el conjunto sal para el hijo izquierdo de la raíz es el conjunto gen para dicho nodo, o 1110000. Este es también el valor del conjunto ent en el nodo **do**. Según las ecuaciones de flujo de datos asociadas con la producción **do** de la figura 10.21, el conjunto ent para la proposición dentro del lazo do se obtiene tomando la unión del conjunto ent 1110000 en el nodo **do** y el conjunto gen 000 1111 en la proposición. La unión produce 111 1111, de modo que todas las definiciones pueden alcanzar el comienzo del cuerpo del lazo. Sin embargo, en el punto justo anterior a la definición d5, el conjunto ent es 0 11 1110, ya que la definición d4 desactiva d1 y d7. El equilibrio de los cálculos ent y sal se deja como ejercicio. 1

**Definiciones de alcance local**

El espacio para la información sobre el flujo de datos se puede cambiar por tiempo, guardando información sólo en ciertos puntos y recalculando la información en los puntos intermedios cuando sea necesario. Los bloques básicos se consideran generalmente como una unidad durante el análisis global de flujo, limitando la atención sólo a aquellos puntos que estén al comienzo delos bloques. Como generalmente hay muchos más puntos que bloques, limitar el esfuerzo a los bloques es un ahorro significativo. Cuando se necesiten, las definiciones de alcance para todos los puntos de un bloque se pueden calcular a partir de las definiciones de alcance para elcomienzo del bloque.

Con más detalle, considérese una secuencia de asignaciones Si S, en un bloque básico B. El comienzo de B es el punto p0, el punto entre las proposiciones Si y Si+1, pi y el final del bloque el punto pn Las definiciones que alcanzan el punto pi se pueden obtener de ent [B] considerando las proposiciones S1 ; S2 ; . . . ; Si. comenzando con SI y aplicando las ecuaciones de flujo dedatos de la figura 10.21 para cascadas de proposiciones. Al inicio, sea D = ent [B]. Cuando se considera Si, se eliminan de D las definiciones desactivadas por Si y se añaden las definiciones generadas por Si. Al final, D está formado por las definiciones que alcanzan pj.

**Cadenas de uso y definición**

A menudo es conveniente almacenar la información de las definiciones de alcance como "cadenas de uso y definición", que son listas, para cada uso de una variable. de todas las definiciones que alcanzan dicho uso. Si un uso de una variable a en el bloque B no viene precedido por ninguna definición no ambigua de a, entonces la cadena de definición y uso para dicho uso de a es el conjunto de definiciones en ent [B] que son definiciones de a. Si hay definiciones no ambiguas de a dentro de B que preceden este uso de a, entonces sólo la última dedichas definiciones de a estará en la cadena de uso y definición, y no se coloca ent [B] en ella. Además, si hay definiciones ambiguas de a, entonces todas aquellas para las que no haya ninguna definición ambigua de a entre ella y el uso de a estarán en la cadena de uso y definición para este uso de a.

**Orden de evaluación**

Las técnicas para conservar espacio durante la evaluación de atributos, estudiadas en el capítulo 5, también se aplican al cálculo de la información sobre el flujo de datos utilizando especificaciones como la de la figura 10.2 1. Específicamente, la única limitación en cuanto al orden de evaluación para los conjuntos de proposiciones gen, desact, ent y sal es el impuesto por las dependencias entre estos conjuntos. Habiendo elegido un orden de evaluación, se puede desocupar el espacio para un conjunto después de que se hayan producido todos sus usos.

Las ecuaciones de flujo de datos de esta sección difieren en un aspecto de las reglas semánticas para atributos del capítulo 5: en el capítulo 5 no se permitían las dependencias circulares entre atributos, pero no se ha visto que las ecuaciones de flujo de datos pueden tener dependencias circulares; por ejemplo, ent [S1] y sal [S1] dependen el uno del otro en 10.8. En el caso del problema de las definiciones de alcance, se pueden reescribir las ecuaciones de flujo de datos para eliminar la circularidad -compárense las ecuaciones no circulares de la Fig. 10.21 con las ecuaciones de l0.8-. Una vez que se obtiene una especificación no circular, se pueden aplicar lastécnicas del capítulo 5 para obtener soluciones eficientes de ecuaciones de flujo de datos.

**Flujo de control general**

El análisis de flujo de datos debe tener en cuenta todos los caminos del control. Si los caminos del control resultan evidentes a partir de la sintaxis, entonces las ecuaciones de flujo de datos se pueden establecer y resolver de manera dirigida por la sintaxis, como en esta sección. En el caso de aquellos programas que pueden contener proposiciones **goto**, o incluso las proposiciones más disciplinadas **break** y **continue**, el enfoque elegido debe modificarse para que tenga en cuenta los caminos reales del control.

Se pueden adoptar varios enfoques. El método iterativo de la siguiente sección trabaja para grafos de flujo arbitrarios. Como los grafos de flujo obtenidos en la presencia de proposiciones **break** y **continue** son reducibles, se pueden considerar sistemáticamente dichas construcciones utilizandolos métodos basados en intervalos, que se estudiará en la sección 10. 10.

Sin embargo, no es necesario abandonar el enfoque dirigido por la sintaxis cuando se permiten proposiciones **break** y **continue**. Antes de terminar esta sección se considera un ejemplo que propone cómo alojar las proposiciones **break**. Estas ideas se desarrollarán en la sección 10. 10.

Ejemplo 10.14. La proposición **break** dentro del lazo **do-while** de la figura 10.24 es equivalente a un salto al final del lazo. ¿Cómo hay que definir entonces el conjunto gen parala siguiente proposición?

**if** e3 **then** a := u2

**else begin** i:= u3; **break end**

Se define el conjunto gen como {d6}, donde d6, es la definición a:=u2, porque d6 es la única definición generada a lo largo de los caminos del control desde el punto inicial al final de la proposición. La definición d7, es decir, i:=u3, se tomará en cuenta al considerar el lazo **do-while** completo.

/\* d1 \*/ i := m-1;

/\* d2 \*/ j := n;

/\* d3 \*/ a := ul;

do

/\* d4 \*/ i :=i+1;

/\* d5 \*/ j := j-1;

if e3 then

/\* d6 \*/ a := u2;

else begin

/\*d7 \*/ i :=u3;

break

end

while e2

Programa que contiene una proposición **break**.

Hay un truco de programación que permite ignorar el salto causado por la proposición **break** mientras se procesan las proposiciones dentro del cuerpo del lazo: se consideran losconjuntos gen y desact para una proposición **break** como el conjunto vacío y U, el conjunto universal de todas las definiciones, respectivamente, como se muestra en la figura 10.25. Los conjuntos restantes gen y desact que se muestran en la figura 10.25 se determinan por medio delas ecuaciones de flujo de datos de la figura 10.21, y el conjunto gen se muestra encima del conjunto desact. Las proposiciones S1 y S2 representan secuencias de asignaciones. Los conjuntosgen y desact en el nodo do quedan por determinar.

No se puede alcanzar el punto final de una secuencia de proposiciones que finalice con una proposición **break**, así que se puede considerar el conjunto gen para la secuencia como 0 y elconjunto desact como U; el resultado seguirá siendo una estimación conservadora de ent y sal. De manera similar, el punto final de la proposición **if** sólo se puede alcanzar a través de la parte **then**, y los conjuntos gen y desact de la figura 10.25 son los mismos que los de su segundo hijo.

Los conjuntos gen y desact en el nodo **do** deben tener en cuenta todos los caminos desde el comienzo hasta el final de la proposición **do**, así que se ven afectados por la proposición **break**. Ahora se calcularán dos conjuntos, G y D, inicialmente vacíos, a medida que se recorre el camino de puntos desde el nodo **do** hasta el nodo **break**. La intuición es que G y D representan las definiciones generadas y desactivadas conforme el control fluye a la proposición **break** desde el comienzo del cuerpo del lazo. Entonces se puede determinar el conjunto gen para la proposición **do-while** tomando la unión de G y el conjunto gen para el cuerpo del lazo, porque el control puede alcanzar el final de **do** desde la proposición **break** o cayendo por el cuerpo del lazo. Por la misma razón, se determina el conjunto desact para el **do** tomando la intersección de D y el conjunto desact del cuerpo del lazo.

Antes de alcanzar el nodo **if** se tiene G == gen [S2] = {d4, d5} y D = desact [S2] = ={d1, d2, d7} En el nodo **if** interesa el caso en que el control fluye a la proposición **break**, de modo que la parte **then** del condicional no afecta a G y D. El siguiente nodo a lo largo del camino de puntos es para una secuencia de proposiciones, así que se calculan nuevos valores de G y D. Escribiendo S3 para la proposición representada por el hijo izquierdo del nodo de secuencia (el etiquetado con d7 ) se utiliza

G := gen [S3] u (G - desact [S3])

D := desact [S3] u (D - gen [S3])

Por tanto, los valores de G y D al llegar a la proposición break son {d5, d7} y {d1. d2, d4}, respectivamente.