Compiladores

Ingeniería Informática, 4º curso

Segundo cuatrimestre:

- Análisis semántico
- Generación de código
- Representación de la información: gestión de la memoria
- Estudio de algunos compiladores

Profesores: **Bernardino Arcay**
Carlos Dafonte
Departamento de Tecnoloxías da Información e as Comunicacións.
Universidade da Coruña

ÍNDICE

5 ANÁLISIS SEMÁNTICO. 1

5.1 Definiciones dirigidas por la sintáxis. 1
5.1.1 Grafos de dependencias. 3

5.2 Esquema de traducción 4

5.3 Comprobaciones en tiempo de compilación. 5
5.3.1 Sistemas de tipos. 6
5.3.2 Una gramática y sus comprobaciones de tipos. 6
5.3.3 Equivalencia de expresiones de tipos. 8
5.3.4 Codificación de tipos. 8
5.3.5 Conversión de tipos. 9
5.3.6 Sobrecarga de funciones y operadores 10

6 GENERACIÓN DE CÓDIGO. 13

6.1 Lenguajes intermedios. 13
6.1.1 Notación Polaca Inversa (RPN) 13
6.1.2 Cuartetos 13
6.1.3 Tercetos. 15

6.2 Generación de código intermedio. 16
6.2.1 Generación de RPN desde expresiones aritméticas. 16
6.2.2 Generación de cuartetos 17

6.3 Generación de código desde lenguaje intermedio. 18
6.3.1 Definición de la máquina objeto. 18
6.3.2 Generación de código desde RPN. 19
6.3.3 Generación de código desde cuartetos. 20

7 OPTIMIZACIÓN DE CÓDIGO. 22

7.1 Algoritmo de Nakata. 24

7.2 Un ejemplo de optimización manual. 28

7.3 Lazos en los grafos de flujo. 28

7.4 Análisis global del flujo de datos. 29
7.4.1 Alcance de definiciones en estructuras de control. 31
7.4.2 Notación vectorial para representar genera y desactiva. 34

7.5 Solución iterativa de las ecuaciones de flujo de datos. 37
7.5.1 Análisis alcance de definiciones. 37
7.5.2 Análisis de expresiones disponibles. 40
7.5.3 Análisis de variables activas. 43

8 ERRORES. 46

8.1 Tipos de errores. 46

8.2 Recuperación de errores léxico-gráficos. 47
8.2.1 Corrección de errores de sustitución. 47
8.2.2 Corrección de errores de borrado. 49
8.2.3 Corrección de errores de inclusión. 49

8.3 Análisis sintáctico en “modo pánico”. 50

9 INTÉRPRETES. 51

9.1 Estructura de un intérprete actual. 51

9.2 Arquitectura “neutral” de Java 53

BIBLIOGRAFÍA.

Aho, A.V.; Sethi, R. ; Ullman, J.D.
"Compiladores: Principios, técnicas y herramientas"
Addison-Wesley, Reading, Massachussetts (1991).

Louden D. K. [2004], Construcción de compiladores. Principios y Práctica, Paraninfo Thomson Learning.

Garrido, A. ; Iñesta J.M. ; Moreno F. ; Pérez J.A. [2004] Diseño de compiladores, Publicaciones Universidad de Alicante.

Sanchis, F.J.; Galán, J.A.
"Compiladores, teoría y construcción"
Ed. Paraninfo (1987).

Aho, A.V.; Ullman, J.D.
"The theory of parsing, translation and compiling", I y II
Prentice-Hall (1972).

Aho, A.V.; Ullman J.D.
"Principles of compiler design"
Addison-Wesley, Reading, Massachussetts.

Hopcroff, J.E. ; Motwani R. ; Ullman, J. D. [2002] Introducción a la teoría de autómatas, lenguajes y computación, Addison-Wesley, Madrid.

Allen I.; Holub
"Compiler design in C"
Prentice-Hall (1991).

Sánchez, G.; Valverde J.A.
"Compiladores e Intérpretes"
Ed. Díaz de Santos (1984).

Sudkamp T.A.
“Languages and machines”
Addison-Wesley.

5 ANÁLISIS SEMÁNTICO.

En esta fase se pretende encontrar errores semánticos; en este capítulo nos centraremos principalmente en la recopilación de información sobre los tipos para posteriormente realizar la generación de código.

Para la realización del análisis semántico se emplea la estructura jerárquica definida durante el análisis sintáctico para reconocer los operadores, operandos de expresiones y proposiciones.

Dentro del análisis semántico, una de las funciones principales es la de verificar si los operadores tienen los operandos del tipo correcto según el lenguaje. Por ejemplo, puede significar un error sumar un número real y un número entero en un lenguaje, sin embargo, en otro, es correcto pero se precisa realizar una conversión.

Las **acciones semánticas** nos van a permitir asociar información a las producciones gramaticales, de forma que incorporamos reglas semánticas e introducimos atributos a los símbolos de la gramática.

Para trabajar con reglas o acciones semánticas se utilizan dos notaciones:

1) Definiciones dirigidas por la sintáxis.
2) Esquemas de traducción.

En general, el metodo consistirá en construir el árbol sintáctico, crear el grafo de dependencias y evaluar las reglas semánticas. Los pasos serían:

CADENA $\rightarrow$ ÁRBOL $\rightarrow$ GRAFO DE DEPEN. $\rightarrow$ EVALUACIÓN REGLAS SEMÁNTICAS

5.1 Definiciones dirigidas por la sintáxis.

Una definición dirigida por la sintáxis es una generalización de una gramática independiente del contexto en la que cada símbolo gramatical tiene un conjunto de atributos asociados (pueden ser sintetizados o heredados).

Forma de una definición dirigida por la sintáxis.

Sea una GIC, G = (N, T, P, S), cada producción $A \rightarrow \alpha$ tiene asociado un conjunto de reglas semánticas de la forma $b = f(c_1, c_2, \ldots, c_k)$, donde f es una función, c_1, c_2, ..., c_k son atributos de los símbolos gramaticales de la producción y b puede ser:
a) Un atributo sintetizado de A (a partir de atributos de la parte derecha).

b) Un atributo heredado de uno de los símbolos gramaticales que está en el lado derecho de la producción (calculado a partir de atributos de la parte izquierda y/o de sus hermanos en la parte derecha).

Se dice que un atributo es **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.

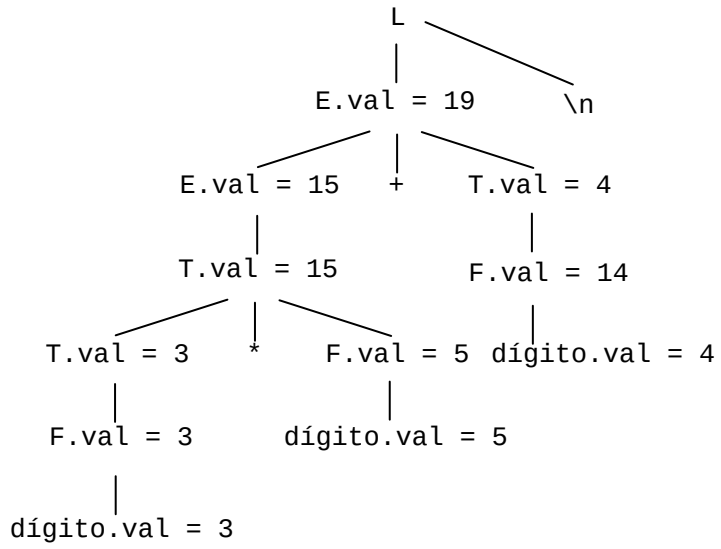
Se dice que un atributo es **heredado** si su valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y/o de los hermanos de dicho nodo.

Una definición dirigida por la sintáxis que usa exclusivamente atributos sintetizados se llama *definición con atributos sintetizados*.

Ejemplo.- Definición con atributos sintetizados.

$L \rightarrow E$ \n	Print (E.val)
$E \rightarrow E_1 + T$	E.val = E_1.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T_1 * F$	T.val = T_1.val x F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow$ dígito	F.val = dígito.val
$T \rightarrow F$	T.val = F.val

Para la entrada "3*5+4\n ", el árbol es:



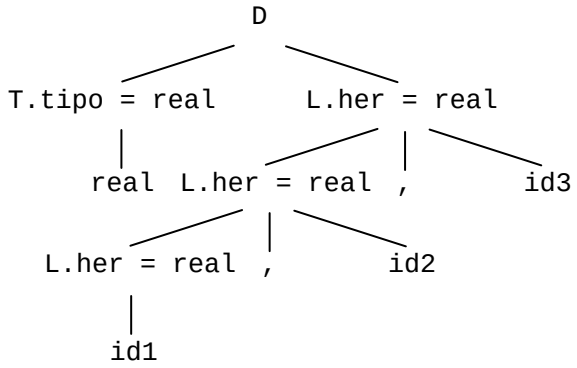
Definición de atributos heredados. Veámoslo con el siguiente ejemplo:

$D \rightarrow TL$	L.her = T.tipo
$T \rightarrow$ **int**	T.tipo = integer
$T \rightarrow$ **real**	T.tipo = real
$L \rightarrow L_1$ **, id**	L_1.her = L.her

	Añadetipo (**id**.entrada, L.her)
L → **id**	Añadetipo (**id**.entrada, L.her)

El procedimiento *añadetipo* añadiría el tipo de cada identificador en su zona de la tabla de símbolos.

A continuación mostramos el árbol sintáctico con anotaciones para la entrada "real id1, id2, id3". Los atributos L.her se obtienen a partir del valor del atributo T.tipo en el hijo izquierdo de la raíz y evaluando después L.her de forma descendente en los tres nodos de L en el subárbol derecho de la raíz. En los tres nodos de L, además, se llama al procedimiento *añadetipo*.



5.1.1 Grafos de dependencias.

Si un atrituto d de un nodo del árbol sintáctico es función de c_1, c_2, ..., c_k ahí tenemos una dependencia, pues tendremos que calcular antes los c_1, c_2, ..., c_k que d. Estas interdependencias entre los atributos sintetizados y/o heredados se pueden representar por un grafo que llamaremos "grafo de dependencias". Es decir, nos va a permitir determinar el orden de evaluación de las acciones semánticas.

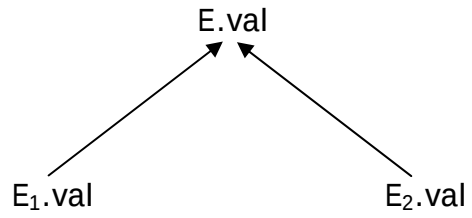
El algoritmo de construcción de este grafo es el siguiente:

for cada nodo *n* en el árbol de análisis sintáctico **do**
    For cada atributo *a* del símbolo gramatical en el nodo *n* **do**
        Construir un nodo en el grafo de dependencias para *a*;
for cada nodo *n* en el árbol de análisis sintáctico **do**
    for cada regla semántica b = f(c_1, c_2, ..., c_k)
    asociada con la producción utilizada en *n* **do**
        for i =1 **to** k **do**
            Construir una arista desde el nodo para c_i hasta *b*;

Ejemplo.-

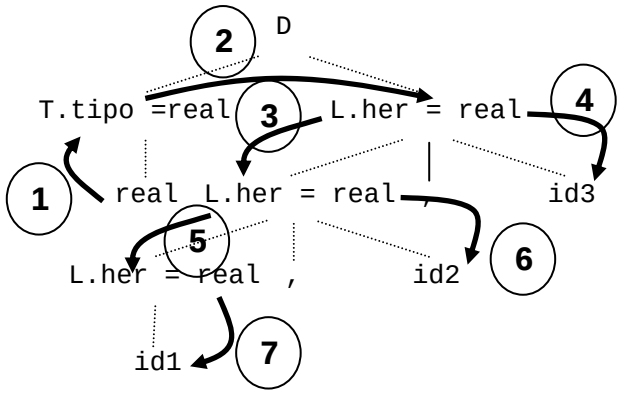
Producción	Regla semántica
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$

Los tres nodos del grafo de dependencias representan los atributos sintetizados E.val, E_1.val y E_2.val en los nodos correspondientes del árbol de análisis.



Llamamos **ordenamiento topológico** de un grafo dirigido a todo ordenamiento m_1, m_2, ..., m_k de los nodos del grafo tal que las aristas vayan desde los nodos que aparecen primero en el ordenamiento a los que aparecen más tarde.

Si tenemos $m_i \rightarrow m_j$, m_i ha de aparecer antes que m_j en el ordenamiento topológico.



5.2 Esquema de traducción

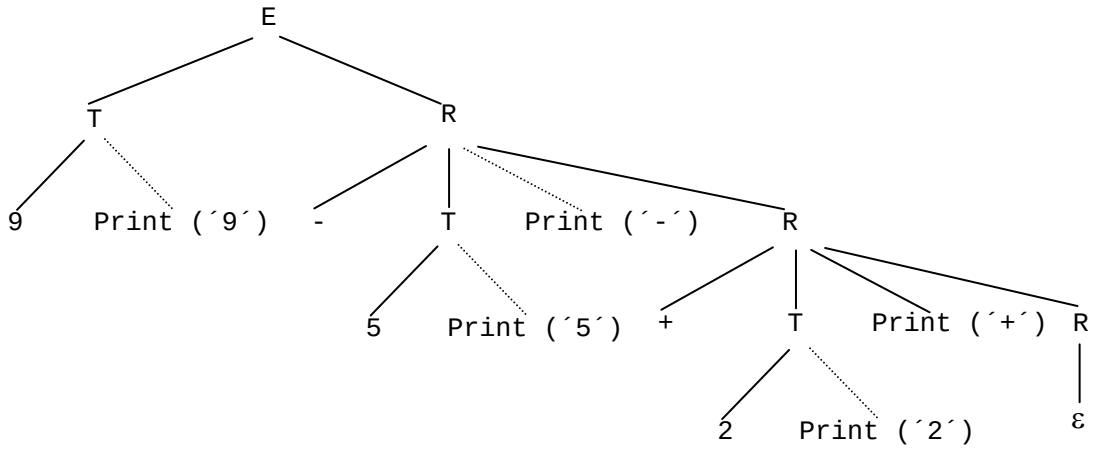
Es otra forma de representar las acciones semánticas, en ella insertamos las acciones semánticas en las reglas entre llaves. También se añaden atributos a los símbolos de la gramática.

Ejemplo.-

$E \rightarrow T\ R$
$R \rightarrow$ **Operador** T {print (**Operador**.lexema)} $R_1 \mid \varepsilon$
$T \rightarrow$ **num** {print (**num**.val)}

Cuando hablamos de **Operador** nos estamos refiriendo a un token que puede ser "+" o "-".

Vamos a ver el árbol de análisis sintáctico para la entrada “9-5+2”, con cada acción semántica asociada como el hijo de su parte izquierda correspondiente:



La salida resultante es: “95-2+”, es decir, la operación en notación postfija.

5.3 Comprobaciones en tiempo de compilación.

Ya hemos comentado que una tarea fundamental de un compilador, en esta etapa, es comprobar que los tipos de datos son los correctos. En relación a esto, algunas de las comprobaciones a realizar en el llamado *análisis estático* (en tiempo de compilación) son las siguientes:

a) **Comprobaciones de tipos**.- El compilador ha de ser capaz de detectar un error cuando se aplica a un operador un operando incompatible (sumar una matriz con un entero, por ejemplo).

b) **Comprobaciones de unicidad**.- Dependerá en gran medida del lenguaje, pero tiene que ver con donde se definen los objetos (por ejemplo, puede ser correcto la definición de variables globales con el mismo nombre que una variable local en un procedimiento o una función, como ocurre en **C**).

c) **Comprobación de operadores sobrecargados**.- Por ejemplo, el operador + significa algo distinto en el caso de 2+3 (operador de suma) que en el caso de “a” + “b” (concatenación).

d) **Polimorfismos en funciones**.- Una función polimórfica es aquella que se puede ejecutar con distintos tipos de argumentos.

Además de éstas, también tenemos comprobaciones del flujo de control; es importante controlar proposiciones que cambian el flujo de control en el programa. Por ejemplo, la proposición *break* en **C** hace que el control salte fuera de la proposición que la engloba (un *switch-case*, *for* o *while*).

5.3.1 Sistemas de tipos.

El comprobador de tipos se basará en las construcciones sintácticas del lenguaje, los tipos definidos y las reglas definidas para asignar los tipos a las construcciones del lenguaje. Todo ello definirá el sistema de tipos.

Por ejemplo, si tenemos la siguiente construcción:

if a > b then a = 25

tendremos que determinar si a y b son comparables por su tipo, tenemos que asignarle a a > b el tipo *boolean*, debemos de saber si a la variable "a" le podemos asignar el valor 25 por su tipo y, finalmente, si a > b es *true*, para lo cual hay que manejar el flujo de control (esto último ya en la ejecución).

Expresiones de tipos: es la forma de denotar el tipo de una construcción de un lenguaje. Una expresión de tipo puede ser un "tipo básico" o formarse aplicando un operador llamado "constructor de tipos" a otras expresiones de tipos. Nosotros utilizaremos la siguiente definición de *expresiones de tipos*:

1) **Tipos básicos**: algunos de ellos son los *integer, boolean, real, char, error* (identificación de un error), *null* (ausencia de valor).

2) **Constructores de tipos**:
   a) Matrices: *array(I,T)*, siendo I el índice y T una expresión de tipo, indica un tipo matriz con índices I y del tipo T.
   b) Productos: T_1 **x** T_2, producto cartesiano del tipo T_1 y T_2.
   c) Registros: Como un producto pero con campos con nombre.
   d) Punteros: *pointer (T)*, es un apuntador a un objeto de tipo T.
   e) Funciones: Las funciones se pueden definir como transformaciones desde un dominio tipo D a un rango tipo R, de la forma D $\rightarrow$ R. Por ejemplo int x int $\rightarrow$ int.

3) **Nombres de tipos**.- Por ejemplo, si en Pascal definimos:
   type enlace = $\uparrow$ nodo;
   type liga = $\uparrow$ nodo;
   var p : $\uparrow$ nodo;
   q : enlace;
   r : liga;

   El problema es saber si son del mismo tipo p, q y r... depende de la implementación pues en la definición original de Pascal no se definió el concepto "tipo idéntico".

5.3.2 Una gramática y sus comprobaciones de tipos.

Para mostar el funcionamiento de los sistemas de comprobación de tipos veamos a continuación una gramática que iremos ampliando paulatinamente a lo largo del presente tema, tanto en número de producciones como en acciones semánticas:

$P \rightarrow D$; E
$D \rightarrow D$; D | **id** : T
$T \rightarrow$ **char** | **integer** | **array [num] of** T | $\uparrow T$
$E \rightarrow$ **literal** | **num** | **id** | E **mod** E | E **[** E **]** | $E\uparrow$

La parte del esquema de traducción que almacena el tipo de un identificador sería la siguiente:

$P \rightarrow D$; E	
$D \rightarrow D$; D	
$D \rightarrow$ **id** : T	{AñadeTipo (**id**.entrada, T.tipo)}
$T \rightarrow$ **char**	{T.tipo := char}
$T \rightarrow$ **integer**	{T.tipo := integer}
$T \rightarrow \uparrow T_1$	{T.tipo := pointer (T_1.tipo)}
$T \rightarrow$ **array [num] of** T_1	{T.tipo := array (1…**num**.val, T_1.tipo)}

Reglas de comprobación de tipos en expresiones**:**

$E \rightarrow$ **literal**	{E.tipo := char}
$E \rightarrow$ **num**	{E.tipo := integer}
$E \rightarrow$ **id**	{E.tipo := BuscaTipo(**id**)}
$E \rightarrow E_1$ **mod** E_2	{E.tipo := IF E_1.tipo = integer AND E_2.tipo = integer THEN integer ELSE error}
$E \rightarrow E_1$ **[** E_2 **]**	{E.tipo := IF E_2.tipo = integer AND E_1.tipo = array (S, T) THEN T ELSE error}
$E \rightarrow E_1\uparrow$	{E.tipo := IF E_1.tipo = pointer(T) THEN T ELSE error}

Reglas de comprobación de tipos en proposiciones**:**

$S \rightarrow$ **id** := E	{S.tipo := IF BuscaTipo(**id**) = E.tipo THEN null ELSE error}
$S \rightarrow$ **if** E **then** S_1	{S.tipo := IF E.tipo = boolean THEN S_1.tipo ELSE error}
$S \rightarrow$ **while** E **do** S_1	{S.tipo := IF E.tipo = boolean THEN S_1.tipo ELSE error}
$S \rightarrow S_1$; S_2	{S.tipo := IF S_1.tipo = null AND S_2.tipo = null THEN null ELSE error}

Reglas de comprobación de tipos en funciones**:**

$E \rightarrow E_1$**(**E_2**)**	{E.tipo := IF E_2.tipo = s AND E_1.tipo = $s \rightarrow t$ THEN t ELSE error}

5.3.3 Equivalencia de expresiones de tipos.

Es importante tener una definición precisa sobre cuando dos expresiones de tipo son equivalentes. Surgen ambigüedades especialmente cuando se asigna un nombre a estas expresiones de tipo.

Para poder implementar en un ordenador la equivalencia de tipos se utiliza la denominada "equivalencia de nombres" y la "equivalencia estructural", es esta última en la que nos vamos a centrar.

Equivalencia estructural de tipos:

Como hemos visto las expresiones de tipos se construyen a partir de tipos básicos y constructores de tipos. Partiendo de estos conceptos decimos que existe una equivalencia estructural de dos expresiones si son el mismo tipo básico o están formadas aplicando el mismo constructor a tipos estructuralmente equivalentes.

Una función válida para saber si dos tipos son estructuralmente equivalentes puede ser la siguiente:

```
Function equival (s, t) : boolean
BEGIN
        IF s y t son del mismo tipo básico
          THEN return TRUE
          ELSE IF s = array (s1, t1) AND t = array (s2, t2)
            THEN return equival (s1, s2) AND equival (t1, t2)
            ELSE IF s = pointer (s1) AND t = pointer (t1)
              THEN return equival (s1, t1)
              ELSE IF s = s1 → s2 and t = t1 → t2
                THEN return  equival (s1, t1) and equival (s2, t2)
                ELSE …
                  …
                   ELSE return FALSE
END.
```

5.3.4 Codificación de tipos.

Consideremos la siguiente codificación para los tipos básicos:

TIPO BÁSICO	CODIFICACIÓN
Bolean	0000
Char	0001
Integer	0010
Real	0011

Si además, tenemos expresiones de tipos con los siguientes constructores:

a) pointer (t), un apuntador a tipo t.

b) freturns (t), que señala una función con argumentos y que devuelve un objeto de tipo t.
c) array (t) que indica una matriz.

Las expresiones de tipos formadas mediante la aplicación de dichos constructores tienen una estructura muy uniforme. Algunos ejemplos son los siguientes:

char
freturns (char)
pointer (freturns(char))
array (pointer(freturns(char)))

Si codificamos los constructores de la siguiente forma:

CONSTRUCTOR	CODIFICACIÓN
pointer	01
array	10
freturns	11

La codificación de las expresiones anteriores será:

CONSTRUCTOR	CODIFICACIÓN
char	…000000 0001
freturns (char)	…000011 0001
pointer (freturns(char))	…000111 0001
array (pointer(freturns(char)))	…100111 0001

Como se puede observar los cuatro bits a la derecha codifican el tipo básico y los colocados inmediatamente a su izquierda el constructor aplicado, más a la izquierda el constructor aplicado a este y así sucesivamente.

5.3.5 Conversión de tipos.

Si tenemos la expresión "a+b" tenemos que comprobar los tipos de a y b, bien para comprobar si existe una incompatibilidad de tipos o bien para realizar una conversión (la forma de sumar un entero y un real no es la misma que la forma de sumar dos enteros).

Las conversiones de tipos pueden ser:

a) **Implícitas o coerciones**.- son las que realiza el propio compilador sin intervención del programador.
b) **Explícitas**.- cuando el programador ha de intervenir para que la conversión se realice. Por ejemplo, las conversiones *(tipo) variable* de **C**.

Las conversiones es preferible realizarlas en tiempo de compilación que en tiempo de ejecución pues se ahorra tiempo, algo que es especialmente sencillo en el caso de las constantes.

Un ejemplo de gramática para la conversión de tipos es la siguiente:

$E \rightarrow$ **num**	{E.tipo := integer}
$E \rightarrow$ **num . num**	{E.tipo := real}
$E \rightarrow$ **id**	{E.tipo := BuscaTipo(**id**)}
$E \rightarrow E_1$ **op** E_2	{E.tipo := IF E_1.tipo = integer AND E_2.tipo = integer THEN integer ELSE IF E_1.tipo = integer AND E_2.tipo = real THEN real ELSE IF E_1.tipo = real AND E_2.tipo = integer THEN real ELSE IF E_1.tipo = real AND E_2.tipo = real THEN real ELSE error}

5.3.6 Sobrecarga de funciones y operadores.

Un operador está sobrecargado cuando tiene distintos significados y acepta distintos tipos de datos. Es decir, podemos tener más de un tipo posible en la operación. Un ejemplo típico es la operación de suma.

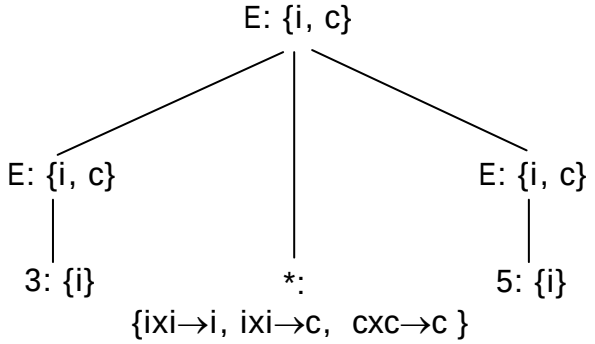
Veamos un ejemplo con una función:

$E \rightarrow E_1$**(**E_2**)**	{E.tipo := IF E_2.tipo = s AND E_1.tipo = $s \rightarrow t$ THEN t ELSE error}

Si existe una sobrecarga, una generalización de esto podría ser la siguiente:

$E' \rightarrow E$	{E´.tipos := E.tipos}
$E \rightarrow$ **id**	{E.tipos := BuscaTipo(**id**)}
$E \rightarrow E_1$**(**E_2**)**	{E.tipos := t / $\exists\ s \rightarrow t \in E_1$.tipos, $s \in E_2$.tipos}

Miremos por ejemplo un árbol para la operación de multiplicación "*", con la expresión "3 * 5":



Como se ve en el árbol, si el único tipo posible para 3 y 5 es integer, entonces el operador * se aplica a un par de enteros (sería integer x integer), sin embargo existen dos posibilidades para el operador *, una devuelve un entero y la otra un complejo, que serán los tipos posibles para la raíz E.

Para reducir el tipo de E a un único tipo (algo obligatorio en algunos lenguajes) se puede implantar una definición dirigida por la sintaxis realizando dos recorridos en profundidad de un árbol sintáctico para una expresión. Durante el primer recorrido, el atributo "tipos" se sintetiza de manera ascendente. Durante el segundo recorrido, el atributo "unico" se propaga de forma descendente (*en cursiva*). Las reglas son:

$S \rightarrow$ **id =** E { *E.unico :=* *IF BuscaTipo(id) nos devuelve un solo tipo*
AND BuscaTipo(id) $\subset$ *E.tipos*
THEN BuscaTipo(id) ELSE error
S.tipo := IF E.unico $\neq$ *error THEN null ELSE error*}

$E \rightarrow$ **id** {E.tipos := BuscaTipo(**id**)}

$E \rightarrow E_1$**(**E_2**)** {E.tipos := t / $\exists$ s $\rightarrow$ t $\in E_1$.tipos, $\exists$s $\in E_2$.tipos
temp := {s / s $\in$ *E_2.tipos AND s* $\rightarrow$ *E.unico* $\in$ *E_1.tipos}*
E_2.unico := IF temp contiene un solo tipo {s}
THEN s ELSE error
E_1.unico := IF temp contiene un solo tipo {s}
THEN s $\rightarrow$ *E.unico ELSE error*}

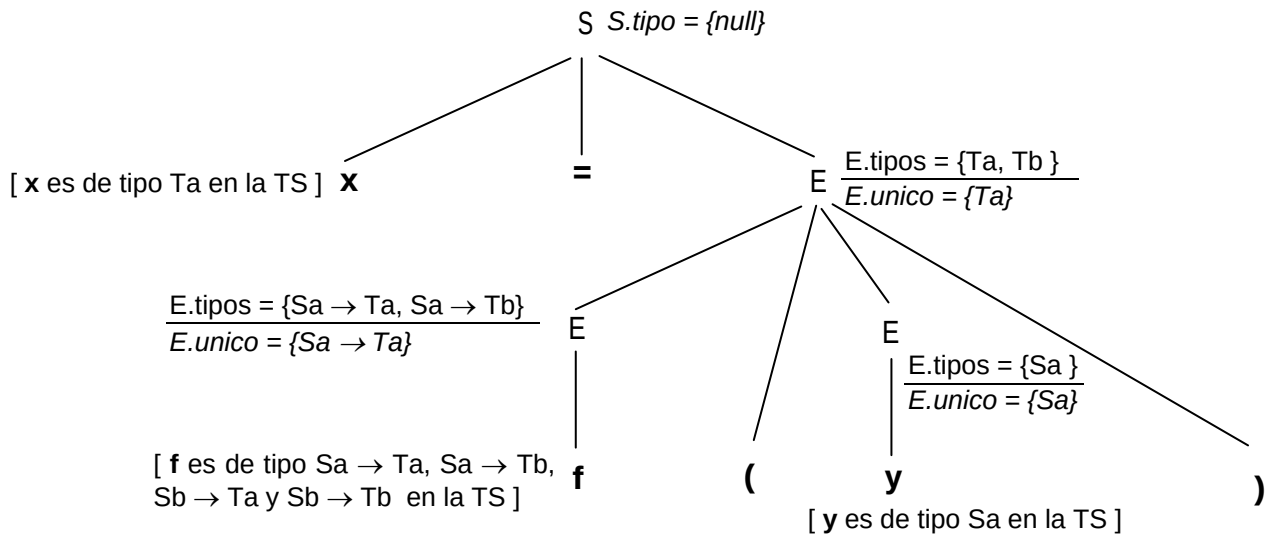
Si tenemos *c = mifuncion(b)*. Nos quedamos con el tipo de *c* (que será el que correspondería a E´ en en análisis descendente de los atributos .unico) y buscamos en el árbol para encontrar una combinación de los tipos de *mifuncion* y de *b* que nos den el tipo que tenemos predefinido para *c*. Si hay más de una combinación, la comprobación no se puede hacer y tendremos un error, y si no hay ninguna tambien se produce un error.

Por ejemplo, si tenemos una frase del lenguaje de la forma: x = f(y), siendo:
x de tipo Ta
fde tipo Sa $\rightarrow$ Ta, Sa $\rightarrow$ Tb, Sb $\rightarrow$ Ta, Sb $\rightarrow$ Tb
y de tipo Sa

De forma ascendente tenemos que:
En $E \rightarrow$ **id** (de la "f") E.tipos = {Sa $\rightarrow$ Ta, Sa $\rightarrow$ Tb, Sb $\rightarrow$ Ta, Sb $\rightarrow$ Tb}
En $E \rightarrow$ **id** (de la "y") E.tipos = {Sa}
En $E \rightarrow E_1(E_2)$, E_1.tipos = {Sa $\rightarrow$ Ta, Sa $\rightarrow$ Tb} y E.tipos={Ta, Tb}
En $S \rightarrow$ **id** = E (**id** aquí es "x", de tipo Ta), E.unico ={Ta} => S.tipo=null

De forma descendente tenemos que E.unico = {Ta}, y con ello podemos reducir a un único tipo cada nodo del árbol (por ejemplo, en nuestro caso, la función será de tipo Sa $\rightarrow$ Ta).

Estos recorridos se pueden ver en la siguiente figura:



6 GENERACIÓN DE CÓDIGO.

La generación de código es una etapa muy relacionada con el procesador (tecnologías CISC y RISC, existencia o no de coprocesador matemático, procesadores vectoriales, sistemas multiprocesador, etc), y precisamente su arquitectura va a influir profundamente en el rendimiento del programa objeto generado.

En esta etapa partimos del árbol resultante del análisis sintáctico y se generará:

1º código intermedio para facilitar la optimización, diseño de fases, etc. Además, si distintos lenguajes general el mismo código intermedio las fases dirigidas a la generación de código máquina pueden unificarse reduciendo los costes de desarrollo.

2º código, normalmente será el código objeto que entiende el procesador aunque podría tratarse también de otro lenguaje distinto (por ejemplo, el GNU Fortran es un traductor de FORTRAN a C).

6.1 Lenguajes intermedios.

6.1.1 Notación Polaca Inversa (RPN)

Esta notación es muy apropiada para analizar y evaluar expresiones aritméticas pero no es demasiado cómoda para utilizarla como lenguaje intermedio de un lenguaje de programación, especialmente para manejar las estructuras de salto.

Una gramática simple para operaciones aritméticas, que genera mediante acciones semánticas código intermedio es la siguiente:

$S \rightarrow S^1 * S^2, S^1 S^2 *$
$S \rightarrow S^1 + S^2, S^1 S^2 +$
$S \rightarrow (S), S$
$S \rightarrow S^1 / S^2, S^1 S^2 /$

6.1.2 Cuartetos

Es la notación más adecuada para representar operaciones binarias. Consta de los siguientes elementos:

(<operador>, <op1>, <op2>, <resultado>)

A / B se escribiría como (/, A, B, t_i), siendo t_i una variable temporal.

Por ejemplo, la siguiente operación:

A + B – C * D ↑ F

Se escribiría con cuartetos de la siguiente forma:

1. (+, A, B, T_1)
2. (↑, D, F, T_2)
3. (*, C, T_2, T_3)
4. (-,T_1, T_3, T_4)

Veremos técnicas para ordenar los cuartetos optimizando las operaciones, incluso llegando a eliminar algunos.

La notación de cuartetos que usaremos será la siguiente:

a) +, -, *, /, ↑.
b) mod (módulo), abs (valor absoluto) y sqr (raíz cuadrada).
c) sin, cos.
d) Asignación (:=, E, , X) que representa X:=E.
e) Entrada / Salida: (READ, , ,X), (WRITE, , , X)
f) Salto absoluto: (JP, n, ,) salta al cuarteto número n.
g) Salto relativo: (JR, n, ,) salta a la posición actual + n.
h) Saltos condicionales:
   - (JZ, n, E,) Si E = 0 salta a n.
   - (JGZ, n, E,) Si E > 0 salta a n.
   - (JLZ, n, E,) Si E < 0 salta a n.
   - (JE, n, X_1, X_2) Si X_1=X_2 salta a n.
   - (JG, n, X_1, X_2) Si X_1>X_2 salta a n.
   - (JL, n, X_1, X_2) Si X_1<X_2 salta a n.

Ejemplo.- Veamos a continuación un bloque de código en Pascal y su correspondiente código en forma de cuartetos:

```
BEGIN
        s:=0;
        i:=0;
        read (n);
100:    i:=i+1;
        read (a);
        s:=s+sqr(a);
        if i<n then goto 100;
        write (s);
END.
```

Código en cuartetos:

1. (:=, 0, , s)
2. (:=, 0, , i)
3. (READ, , , n)
4. (+, i, 1, t_1)
5. (:=,t_1, , i)
6. (READ, , , a)
7. (sqr, a, , t_2)
8. (+, s, t_2, t_3)
9. (:=, t_3, , s)
10. (JL, 4, i, n)
11. (WRITE, , , s)

6.1.3 Tercetos.

La notación de cuartetos es una de las más utilizadas como lenguaje intermedio, sin embargo tiene el inconveniente de ocupar mucho espacio en memoria debido a la utilización de gran cantidad de variables auxiliares.

Los tercetos ahorran espacio eliminando el campo de resultado, quedando implícito, asociado a dicho terceto. El formato de un terceto es el siguiente:

(<operador>, <op1>, <op2>)

La expresión "A+B/C", expresada en forma de tercetos, se representa como:

1. (/, B, C)
2. (+, A, [1])

Con el [1] hacemos referencia al resultado obtenido del primer terceto.

Debido a que el resultado de un terceto está implícito, al realizar una reordenación de los tercetos, por ejemplo en una optimización, hace que estos sean difíciles de manejar (hay que cambiar las referencias). Por ello, se ha definido una modificación denominada "tercetos Indirectos".

En el caso de tercetos indirectos disponemos de dos estructuras: una son los tercetos tal cual teníamos anteriormente y otra es el denominado "vector secuencia", que indica la secuencia de ejecución de los tercetos.

Con este esquema, al reordenar la ejecución no es preciso cambiar en los propios tercetos las referencias a resultados de otros tercetos, y además si tenemos tercetos repetidos en la ejecución, éstos se indicarán solamente en el vector de secuencia.

Ejemplo.- Vamos a escribir en formato de tercetos el mismo código del apartado anterior.

1. (:=, 0, s)
2. (:=, 0, i)

3. (READ, , n)
4. (+, i, 1)
5. (:=, [4], i)
6. (READ, , a)
7. (sqr, , a)
8. (+, s, (7))
9. (:=, (8), s)
10. (-, i, n)
11. (JLZ, 4, [10])
12. (WRITE, , s)

Nótese que aquí no tenemos instrucciones JE, JG y JL, por el número de operandos de los tercetos.

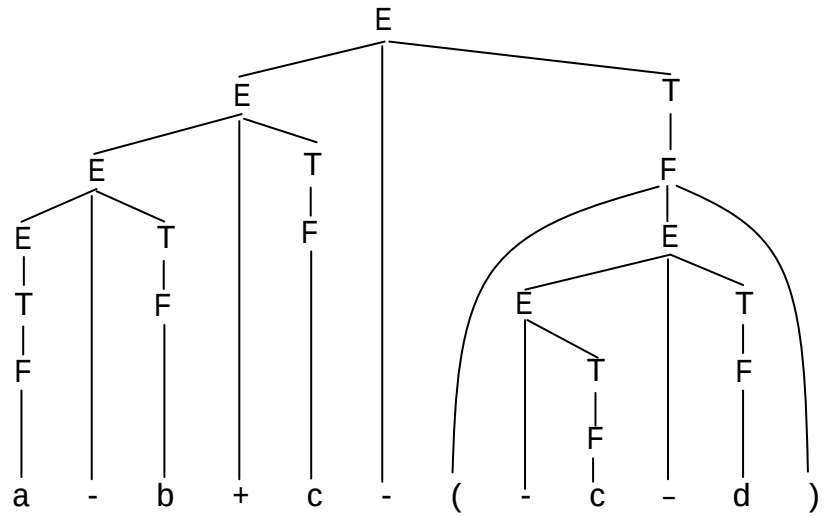
6.2 Generación de código intermedio.

6.2.1 Generación de RPN desde expresiones aritméticas.

Para la generación de la pila con la expresión en formato RPN, utilizaremos las acciones semánticas durante el parsing. A continuación vemos un ejemplo de gramática con reglas semánticas para realizar esta generación (con P(p) indicamos la cabeza de la pila):

REGLAS BNF	ACCIONES SEMÁNTICAS
S → E	
E → T	
E → E + T	P(p) = ´+´, p = p+1
E → E – T	P(p) = ´-´, p = p+1
E → - T	P(p) =´@´, p = p+1
T → F	
T → T * F	P(p) = ´*´, p = p+1
T → T / F	P(p) = ´/´, p = p+1
F → id	P(p) = id, p = p+1
F → (E)	

Ejemplo.- Vamos a ver el árbol para “a-b+c-(-c-d)”



Haciendo el recorrido de "izquierda-derecha-centro" obtenemos el estado de la pila, que será: "ab-c+c@d--".

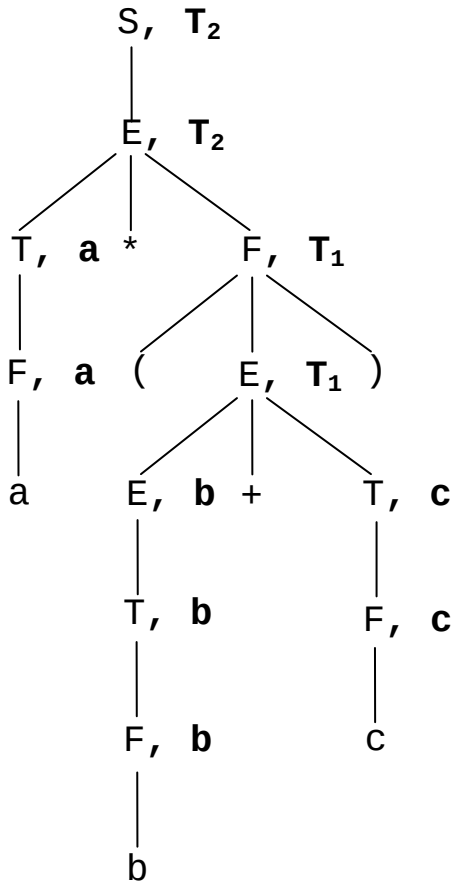
6.2.2 Generación de cuartetos.

A la hora de generar los cuartetos a partir del árbol sintáctico, hay que tener en cuenta el problema del arrastre de la información semántica, para eso utilizamos la notación ".sem". Es decir, tendremos reglas semánticas de arrastre de atributos, así como reglas de generación de código. Además, crearemos variables temporales Ti cuando tengamos resultados de operaciones.

Veamos ahora una gramática y sus reglas asociadas:

$S \rightarrow E$	S.sem = E.sem
$E \rightarrow T$	E.sem = T.sem
$E_1 \rightarrow E_2 + T$	i=i+1; E_1.sem=Ti; Genera ["(´+´, E_2.sem, T.sem, E_1.sem)"]
$E_1 \rightarrow E_2 - T$	i=i+1; E_1.sem=T_i; Genera ["(´-´, E_2.sem, T.sem, E_1.sem)"]
$E \rightarrow -T$	i=i+1; E_1.sem=T_i; Genera ["(´@´, ∅, T.sem, E_1.sem)"]
$T \rightarrow F$	T.sem = F.sem
$T_1 \rightarrow T_2 * F$	i=i+1; T_1.sem=T_i; Genera["(´*´, T_2.sem, F.sem, T_1.sem)"]
$T_1 \rightarrow T_2 / F$	i=i+1; T_1.sem=T_i; Genera["(´/´, T_2.sem, F.sem, T_1.sem)"]
$F \rightarrow id$	F.sem = id.sem
$F \rightarrow (E)$	F.sem = E.sem

Vamos a ver el árbol para “a*(b+c)”



6.3 Generación de código desde lenguaje intermedio.

6.3.1 Definición de la máquina objeto.

En este apartado vamos a definir una máquina objeto para todos los ejemplos que veamos.

Nuestra máquina dispondrá de n registros: R0, R1, ..., Rn. Hay que tener en cuenta que cualquier operación realizada sobre los registros es mucho más rápida que si se realiza sobre datos en memoria.

Las operaciones máquina serán de la forma: OP fuente, destino. Vamos a suponer que las palabras son de 4 bytes.

Las intrucciones válidas para nuestra máquina serán:

MOV fuente, destino	Llevar la información de la fuente al destino.
ADD fuente, destino	Hace fuente + destino y el resultado en destino.
SUB fuente, destino	Hace destino – fuente y el resultado en destino.
MUL fuente, destino	Hace fuente * destino y el resultado en destino.
DIV fuente, destino	Hace destino / fuente y el resultado en destino.

GOTO posición	Saltos en el programa.
HALT	Parada de programa.
CALL	Llamada a subrutina.
RETURN	Retorno de subrutina.
CMP fuente, destino	Comparar fuente con destino.
CJE posición	Salto a posición si tras CMP fuente = destino.
CJL posición	Salto a posición si tras CMP fuente < destino.
CJG posición	Salto a posición si tras CMP fuente > destino.

Modos de direccionamiento.

MODO	FORMA	DIRECCIÓN	COSTO
Absoluto	M	M	1
Registro	Ri	No dirección, valor Ri	0
Indexado	C(R)	C + Contenido(R)	1
Registro indirecto	*R	Contenido(R)	0
Indexado indirecto	*C(R)	Contenido(C + Contenido(R))	1

Ejemplo.-

MOV R0, M	Copia el contenido de R0 a la posición de memoria M.
MOV 4(R0), M	Copia el contenido de la dirección (4 + Contenido(R0)) a la dirección M.
MOVE *4(R0), M	Copia el contenido de la dirección de memoria almacenada en la dirección (4 + Contenido(R0)) a la dirección M.
MOVE #325h, M	Copia el valor 325h a la dirección de memoria M.

6.3.2 Generación de código desde RPN.

El mecanismo es el siguiente:

a) Si llega un símbolo se almacena en la pila.
b) Si llega un operador se realiza la operación, se sacan los operandos de la pila, se genera el código almacenando el resultado de la operación dentro de la pila.

Veamos el ejemplo con la sentencia "A*B+C-D*E", que en RPN es "AB*C+DE*-", suponiendo que sólo tenemos un registro o acumulador, el R0:

Pila antes	Símbolo actual	Resto sentencia	Acción	Código generado
$	A	B*C+DE*-$	Apilar A	
$A	B	C+DE*-$	Apilar B	

$AB	*	C+DE*-$	Generar	MOV A, R0
				MUL B, R0
				MOV R0, T_1

T_1	C	+DE*-$	Apilar C	
T_1C	+	DE*-$	Generar	MOV T_1, R0 ADD C, R0

				MOV R0, T_2
$\$T_2$	D	E*-$	Apilar D	
$\$T_2$D	E	*-$	Apilar E	

$\$T_2$DE	*	-$	Generar	MOV D, R0
				MUL E, R0
				MOV R0, T_3

$\$T_2T_3$	-	$	Generar	MOV T_2, R0 SUB T_3, R0
$	$		Fin	

Como se puede observar muy fácilmente, el código generado no es óptimo pues el registro se vacía y se vuelve a llenar con el mismo valor repetidas veces. Para solucionar este problema vamos a utilizar un símbolo (**Acc**) que introduciremos en la pila cuando esta está usada; si en algún momento se intenta introducir en la pila dos **Acc**, significará que tenemos que descargarlo en memoria. Veamos el funcionamiento con el ejemplo anterior.

Pila antes	Símbolo actual	Resto sentencia	Acción	Código generado
$	A	B*C+DE*-$	Apilar A	
$A	B	C+DE*-$	Apilar B	
$AB	*	C+DE*-$	Generar	MOV A, R0 MUL B, R0
$**Acc**	C	+DE*-$	Apilar C	
$**Acc**C	+	DE*-$	Generar	ADD C, R0
$**Acc**	D	E*-$	Apilar D	
$**Acc**D	E	*-$	Apilar E	

$**Acc**DE	*	-$	Generar	MOV R0, T_1
				MOV D, R0
				MUL E, R0

$ T_1**Acc**	-	$	Generar	MOV R0, T_2
				MOV T_1, R0
				SUB T_2, R0

$	$		Fin	

6.3.3 Generación de código desde cuartetos.

En general, a partir de un cuarteto genérico:

(OP, Op1, Op2, Resultado)

obtendremos el siguiente código genérico:

MOV Op1, R0
OP Op2, R0
MOV R0, Resultado (nota: este Resultado será un T_i)

Para manejar el acumulador, y previamente a ejecutar la operación OP, actuaremos de la siguiente forma, según las posibilidades:

a) Acumulador vacío: Genera MOV Op1, R0 y luego OP Op2, R0
b) Acumulador lleno con el primer operando: Genera OP Op2, R0
c) Acumulador lleno con el segundo operando:
   - Si la operación es conmutativa: Genera OP Op1, R0
   - Si la operación no es conmutativa: Genera
     ```
     MOV R0, Ti
     MOV Op1, R0
     OP Ti, R0
     ```
d) Acumulador lleno con algo útil, que no es el primero ni el segundo operando: Genera
   ```
   MOV R0, Ti
   MOV Op1, R0
   OP Op2, R0
   ```

7 OPTIMIZACIÓN DE CÓDIGO.

En este capítulo trataremos de optimizar tanto velocidad de ejecución como memoria ocupada por el programa. En cualquier caso, los grandes cambios en el hardware que está apareciendo en al actualidad pueden hacer que muchos de los conceptos que veamos cambien.

En este tema trataremos distintos puntos de vista de la optimización. Trataremos de analizar el flujo de datos. Por ejemplo, si tenemos las siguientes instrucciones en nuestro programa:

```
FOR I=1 TO 100
        {
        …
        a = b + c
        …
        }
```

La operación a = b + c podría realizarse fuera del bucle en el caso de que "a" no cambie. De esta forma aceleraríamos la ejecución del programa.

También veremos algoritmos para optimizar la ejecución de operaciones aritméticas.

Para los ejemplos de este capítulo utilizaremos código de tres direcciones.

Resumiendo, en este capítulo trataremos los temas:

1. **Reducción de operaciones**:

Si tenemos la operación A=4x3+2 dentro de un bucle, entonces lo más lógico será realizar la operación fuera y ese valor será válido para todo el bucle.

2. **Reacondicionamiento de instrucciones**:

La operación: A = B*C*(D+E), podríamos generar el código de las siguientes formas:

Forma 1	Forma 2
MOV B, R0	MOV D, R0
MUL C, R0	ADD E, R0
MOV R0, T1	MUL B, R0
MOV D, R0	MUL C, R0
ADD E, R0	
MUL T1, R0	

Si lo hacemos de la forma 2 obtenemos un código más pequeño y que ocupa menos memoria (no utilizamos ningún Ti).

En general veremos que reordenando las operaciones se optimiza el código. Esto lo haremos con el algoritmo de Nakata.

3. **Eliminación de redundancias**:

Veremos el problema de las redundancias aplicado principalmente a las matrices. Veamos el siguiente ejemplo.-

Definición de la matriz:
a : array [1..4; 1..6; 1..8] of integer

Instrucción dentro de un programa:
a[i, j, 5] := a[i ,j ,6] + a[i ,j ,4] + a[i ,j ,3]

Hay que tener en cuenta que las matrices, al almacenarse en memoria, utilizan direcciones de memoria consecutivas. Vamos a suponer que nuestro compilador almacena los datos en memoria en el mismo orden que aparecen los índices.

Vamos a ver la dirección de memoria de cada uno de los componentes "matriz" de la instrucción se calcularán sumando a la dirección del primero [1, 1, 1] diferentes valores:

a[i, j, 5] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 5 – 1
a[i, j, 6] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 6 – 1
a[i, j, 4] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 4 – 1
a[i, j, 3] dirección: [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8 + 3 – 1

Por lo tanto, vemos que hay una parte común en el cálculo de las direcciones de los elementos de la matriz utilizada en la instrucción (llamémosle K), podríamos calcularlo primero y luego añadir la parte no común. Sería de la siguente forma:

K = [1, 1, 1] + (i-1) * 6 * 8 + (j-1) * 8

a[i, j, 5] dirección: K + 5 – 1
a[i, j, 6] dirección: K + 6 – 1
a[i, j, 4] dirección: K + 4 – 1
a[i, j, 6] dirección: K + 3 – 1

Realizar esto es especialmente importante en el caso de que la instrucción se encontrara en un bucle pues ahorraríamos mucho tiempo de ejecución.

4. **Reducción de potencia**.

El cálculo $a = b^2$ puede implementarse de dos formas distintas:

a) a = b ^ 2
b) a = b * b

Para la máquina, realizar el cálculo es distinto pues normalmente ejecutar la opción (a) es un trabajo "software" y la opción (b) habitualmente se realiza por "hardware", y por lo tanto es más rápido.

7.1 Algoritmo de Nakata.

Fue diseñado originalmente en el año 1964 (Anderson) y fue evolucionando hasta 1967 por los trabajos de Nakata. El objetivo es la optimización del uso de los registros minimizando el uso de variables temporales.

Dada una expresión aritmética, primero construímos el árbol (mediante el análisis sintáctico) donde los nodos son los operadores y las hojas son las variables o identificadores. Luego se etiqueta cada arista del árbol con un factor de peso, que inicialmente era el nº de acumuladores que necesitaba esa expresión, aunque el que veremos aquí llevará un factor relativo.

Partiendo de la raíz se toma aquella rama del árbol o subárbol que quede por analizar con el factor de peso más grande; en caso de igualdad se toma la rama de la derecha y continuaría hasta llegar al final de la expresión.

Algoritmo de etiquetado:

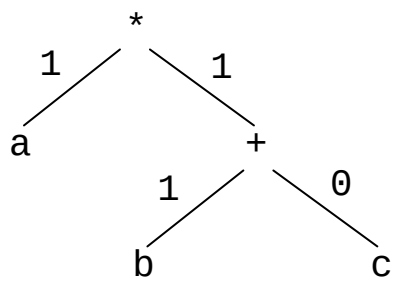
IF n es una hoja THEN
    IF n es el hijo más a la derecha de su padre THEN
        etiqueta (n) = 0
    ELSE
        etiqueta (n) = 1
ELSE
    Sean n_1, n_2, ..., n_k los hijos de n ordenados por etiqueta, de modo que etiqueta (n_1) ≥ etiqueta(n_2) ≥ ... ≥ etiqueta (n_k), hacemos:

$$\text{etiqueta}(n) = \max_{1 \le i \le k} (\text{etiqueta}(n_i) + i - 1)$$

NOTA: En caso de tratarse de un árbol binario, si los hijos tienen igual peso, el padre tendrá ese mismo peso + 1, si los hijos tienen distinto peso, el padre tendrá el peso del mayor.

Ejemplo.- a * (b + c)

El árbol de Nakata es:



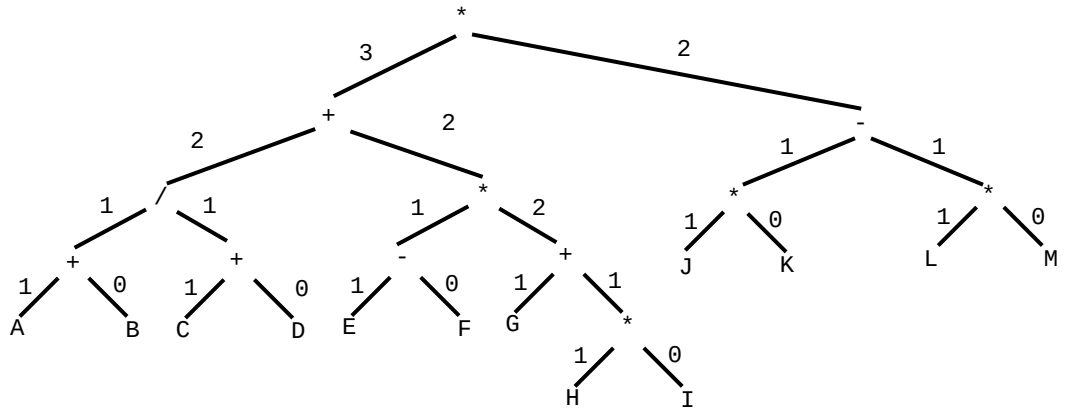
Haciendo el recorrido por Nakata, la expresión queda: (b + c) * a

Veamos ahora el código que generaría la expresión original y la expresión reordenada por Nakata:

a * (b + c)	(b + c) * a
MOV a, R0	MOV b, R0
MOV R0, T1	ADD c, R0
MOV b, R0	MUL a, R0
ADD c, R0	MOV R0, T1
MUL T1, R0	
MOV R0, T2	

Se puede observar como la operación generada por Nakata precisa menos código y menos variables temporales para su ejecución.

Ejemplo.- ((A + B) / (C + D) + (E - F) * (G + H * I)) * (J * K – L * M)



Recorremos el árbol de arriba hacia abajo, siguiendo la rama con mayor peso y obtenemos la siguiente expresión:

((((H * I) + G) * (E – F)) + ((C + D) / (A + B))) * ((L * M) – (J * K))

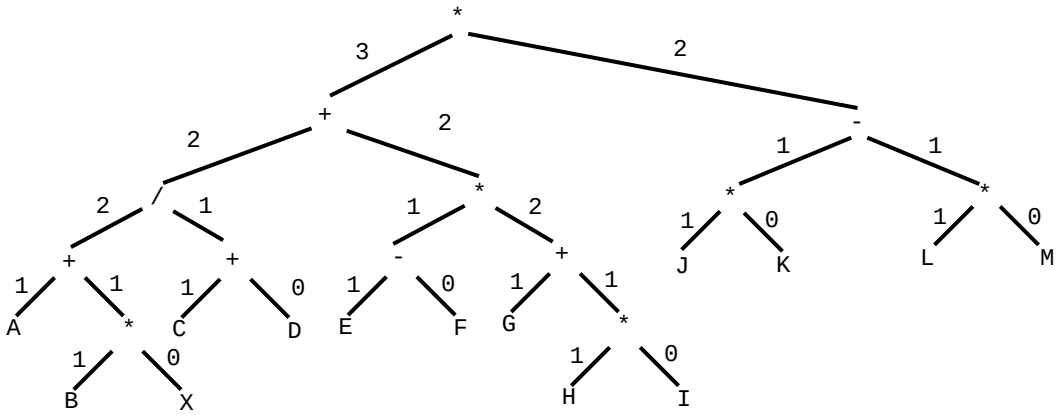
Nakata no tiene en cuenta la conmutatividad o no conmutatividad de las operaciones y nos ha cambiado una operación de división y una resta. Sin embargo, si generamos código suponiendo un acumulador, en este caso funciona por la definición que hemos hecho de nuestra máquina objeto:

MOV H, R0	MOV C, R0	MOV L, R0
MUL I, R0	ADD D, R0	MUL M, R0
ADD G, R0	MOV R0, T3	MOV R0, T5
MOV R0, T1	MOV A, R0	MOV J, R0
MOV E, R0	ADD B, R0	MUL K, R0
SUB F, R0	DIV T3, R0	SUB T5, R0
MUL T1, R0	ADD T2, R0	MUL T4, R0
MOV R0, T2	MOV R0, T4	MOV R0, T6

Ejemplo.- Vamos a hacer una modificación en la operación, de la forma siguiente:

((A + (B * X)) / (C + D) + (E - F) * (G + H * I)) * (J * K – L * M)

El árbol quedaría de la siguiente forma:



Recorremos el árbol de arriba hacia abajo, siguiendo la rama con mayor peso y obtenemos la siguiente expresión:

((((H * I) + G) * (E – F)) + (((B * X) + A) /(C + D))) * ((L * M) – (J * K))

Vemos que ahora ha realizado la división de forma “correcta”, sin embargo, al generar código con la máquina objeto que hemos definido, tenemos que intercambiar los operandos en la división. Veamos el código:

MOV H, R0	MOV B, R0	MOV L, R0
MUL I, R0	MUL X, R0	MUL M, R0
ADD G, R0	ADD A, R0	MOV R0, T6
MOV R0, T1	MOV R0, T3	MOV J, R0
MOV E, R0	MOV C, R0	MUL K, R0
SUB F, R0	ADD D , R0	SUB T6, R0
MUL T1, R0	**MOV R0, T4**	MUL T5, R0
MOV R0, T2	**MOV T3, R0**	MOV R0, T7
	DIV T4, R0	
	ADD T2, R0	
	MOV R0, T5	

NOTA: Las operaciones que están en negrita son las que nos permiten realizar la operación de división de forma correcta. Al hacer el recorrido del árbol sintáctico, llegados a “/” tiene que darse cuenta de que no es conmutativa e invertir los operandos para realizarlo correctamente.

Ejercicio.- Vamos a optimizar el siguiente código:

```
FOR I=L0 TO L1 DO BEGIN
        FOR J=L2 TO L3 DO BEGIN
                M[I] =X1 * X2 * (X3+X4)
                N[I,J] = M[I] ** 2 * D
        END;
END;
```

Vemos que una gran parte del código se puede ejecutar fuera de los bucles, o por lo menos una gran parte. Además, podemos optimizar por Nakata los cálculos y transformar el "elevado al cuadrado" por una simple multiplicación. Nos quedaría los siguiente:

```
K1 = (X3 + X4) * X1 * X2
K2 = K1 * K1 * D
FOR I=L0 TO L1 DO BEGIN
        M[I] = K1
        FOR J=L2 TO L3 DO BEGIN
                M[I,J] = K2
        END;
END;
```

Vamos a generar código en cuartetos:

1. (+, X3, X4, T1)
2. (*, T1, X1, T2)
3. (*, T2, X2, T3)
4. (=, T3, , K1)
5. (*, K1, K1, T4)
6. (*, T4, D, T5)
7. (=, T5, , K2)
8. (=, L0, , I)
9. (JL, 20, L1, I)
10. (=, K1, , M[I])
11. (=, L2, , J)
12. (JL, 17, L3, J)
13. (=, K2, , M[I,J])
14. (+, J, 1, T7)
15. (=, T7, , J)
16. (JP, 12, ,)
17. (+, i, 1, T8)
18. (=, T8, , I)
19. (JP, 9, ,)
20. (END, , ,)

NOTA: Es preciso hacer notar que no hemos generado el código de forma extricta pues no hemos calculado las direcciones de la matrices. Además, vemos que, en los bucles con cuartetos, hemos realizado la comprobación al principio del bucle "FOR", con lo que, un FOR I=2 TO 1 nunca se ejecutaría.

7.2 Un ejemplo de optimización manual.

--- Aho-Ullman Pag 606-616 ---

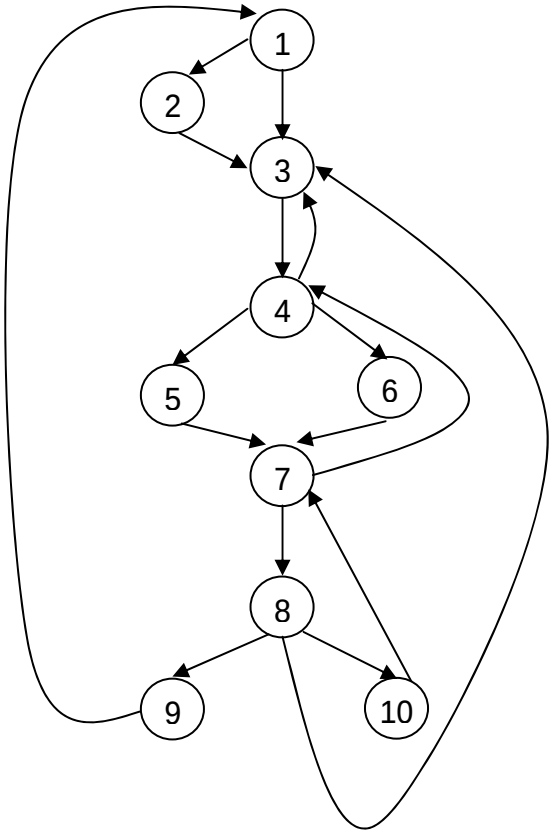
7.3 Lazos en los grafos de flujo.

Vamos a comenzar con algunas definiciones necesarias para este apartado.

Definición: decimos que un nodo d de un grafo de flujo **domina** al nodo n (escribimos "d domina n") si todo camino desde el nodo inicial del grafo de flujo a n pasa por d.

NOTA: hay que tener en cuenta que todo nodo se domina a sí mismo y que el nodo inicial domina a todos.

Ejemplo.-

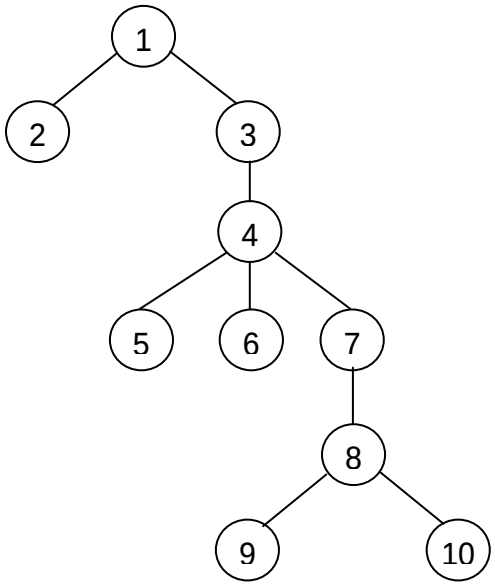


Aquí podemos ver como:

El nodo 1 domina a todos los demás nodos.
El nodo 2 sólo se domina a sí mismo.
El nodo 3 domina a todos excepto al 1 y el 2.
El nodo 4 domina a todos menos al 1, 2 y el 3.
El nodo 5 y el nodo 6 sólo se dominan a ellos mismos.
El nodo 7 domina al 7, 8, 9 y 10
El nodo 8 domina al 8, 9 y 10.

El nodo 9 y el nodo 10 sólo se dominan a sí mismo.

Vamos a ver ahora el denominado "árbol de dominación", que es la representación gráfica de la información anterior.



Una de las aplicaciones principales de la información sobre dominadores es determinar los lazos de un grafo de flujo. Los lazos tienen dos propiedades importantes:

1. Un lazo tendrá un solo punto de entrada (encabezamiento). Este punto de entrada domina a todos los nodos dentro del lazo, será su única entrada.
2. Debe existir al menos una forma de iterar el lazo, es decir, por lo menos existirá un camino hacia el encabezamiento.

Para localizar los lazos en un grafo de flujo, un método es localizar aristas cuyas cabezas dominen a sus colas. Esto es, si tenemos una arista de la forma: $a \rightarrow b$ (b es la cabeza y a es la cola), si sabemos que b domina a entonces significa que ahí hay un lazo.

En el grafo anterior podemos ver que: Tenemos una arista de 7 a 4 y 4 domina 7, de la misma forma, hay una arista de 10 a 7 y 7 domina 10. Los otros lazos que nos encontramos aparece con las aristas de 4 a 3, de 8 a 3 y de 9 a 1.

7.4 Análisis global del flujo de datos.

Para realizar el análisis del flujo de datos y, por lo tanto, realizar una buena optimización, precisamos tener información disponible sobre las variables, expresiones, etc, que será el punto de partida de la optimización.

Nos centraremos especialmente en los bucles, un punto en donde las optimizaciones suelen ser importantes.

Con el análisis de flujo de datos conseguiremos eliminar código inactivo, obtener subexpresiones comunes, etc.

La información del flujo de datos se obtendrá resolviendo ecuaciones que relacionan la información en varios puntos de un programa. Veamos la siguiente ecuación:

$$sal[S] = gen[S] \cup (ent[S] - desact[S])$$

Dependiendo de lo que pretendamos optimizar, variarán las nociones de "generar" y "desactivar".

Es perciso hacer notar que el análisis se realiza habitualmente a nivel de bloque y no de proposición, cuando hablamos de sal[S] estamos hablando de un punto final único, algo que sí existe en los bloques.

Hay casos especiales como son los punteros y las matrices, los cuales hacen más complicado el análisis. También nos pueden complicar el análisis las llamadas a procedimientos. Por ejemplo:

```
x = 5
…
call función(x)
…
x = 6
```

Cuando llamamos a la función puede haber ocurrido que el valor de x cambie.

En cualquier programa consideraremos que existen puntos entre 2 proposiciones (cada proposición tiene, al menos, un punto antes y después).

Un **camino** de un punto p_1 a un punto p_n es una secuencia de puntos p_1, p_2,…, p_{n-1}, de forma que p_i es que punto que precede a una proposición y p_{i+1} el que la sigue en el mismo bloque, o bien p_i es el final de un bloque y p_{i+1} es el comienzo del bloque siguiente.

Llamamos **definición de una variable x** a una proposición que asigna un valor a **x**, o puede asignarlo.

Serán *definiciones no ambiguas*: x=7, scanf("%d", fpepe), lectura de fichero, etc. Serán *ambiguas* las llamadas a procedimientos con x como parámetro (por dirección, no por valor) o sin la variable x como parámetro, pero está en su alcance.

Llamamos **alcance de una definición d a un punto p**, si existe un camino desde el siguiente punto a d hasta p tal que la definición no se desactive.

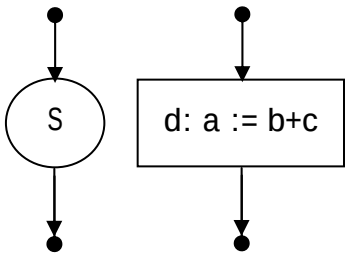
Decimos que una **definición se desactiva** si durante el camino a otro punto se realiza otra definición del mismo elemento.

7.4.1 Alcance de definiciones en estructuras de control.

Para los ejemplos de este apartado utilizaremos la siguiente sintáxis:

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

Vamos a ver las definiciones inductivas, o dirigidas por la sintaxis, de los conjuntos ent[S], sal[S] y genera[S] y desact[S] para las diferentes estructuras:

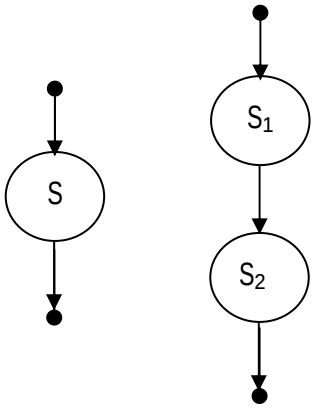


gen[S] = {d}
desact [S] = D_a – {d}

sal[S] = gen[S] ∪ (ent[S] – desact[S])

Aquí estamos diciendo que lo que se genera en esa proposición es la definición que hemos llamado "d". Lo que se desactiva son todas la definiciones que haya en el programa de la variable "a" menos la de la propia proposición.

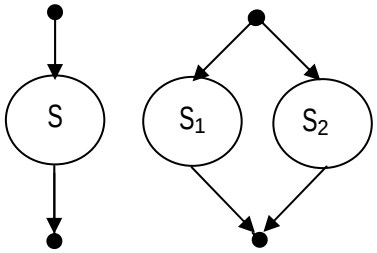
Lo que sale de S es lo que generamos dentro (la definición de "a") unido a todo lo que entra menos lo que desactivamos.



gen [S] = gen[S_2] ∪ (gen[S_1] – desact[S_2])
desact[S] = desact[S_2] ∪ (desact[S_1] – gen[S_2])

ent[S_1] = ent[S]
ent[S_2] = sal[S_1]
sal[S] = sal[S_2]

Lo que se genera en S_2 ya es lo que genera S, además de lo que genere S_1, siempre y cuando no sea desactivado por S_2. Idem con la desactivación.

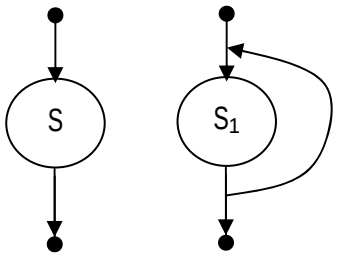


gen [S] = gen[S_1] ∪ gen[S_2]
desact[S] = desact[S_1] ∩ desact[S_2]

ent[S_1] = ent[S]
ent[S_2] = ent[S]
sal[S] = sal[S_1] ∪ sal[S_2]

Aquí utilizamos una política conservadora, asumimos todo lo de S_1 y S_2 (cuando realmente se ejecuta uno de los dos) y en la desactivación, sólo consideramos que se ha desactivado lo que realmente se desactiva en los dos

(pase por el que pase). Esto es correcto si pretendemos ver, por ejemplo, el rango de valores que puede tomar una variable. Si vemos que todas las definiciones en un punto determinado después del "if" son siempre x=9 (por las dos ramas), podemos utilizar 9 en lugar de x. Para este tipo de optimizaciones no hacerlo así nos produciría optimizaciones incorrectas. Si lo que pretendemos es utilizar el valor de la asignación hacerlo así no sería correcto pues "es posible esa rama no se haya ejecutado".



$gen[S] = gen[S_1]$
$desact\ [S] = desact[S_1]$

$ent[S_1] = ent[S] \cup sal[S_1]$
$sal[S] = sal[S_1]$

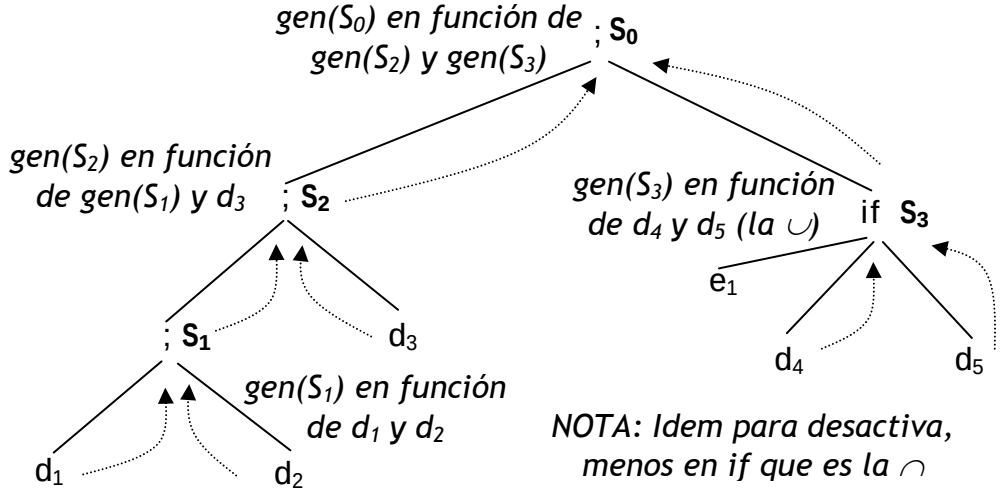
Aquí, la existencia del lazo no afecta a lo que se genera o desactiva en S, así como su salida.

Aquí puede observarse como la ent[S] no es la misma que la ent[S1], y mediante la ecuación anterior, o sea $ent[S_1] = ent[S] \cup sal[S_1]$, no se puede calcular la entrada sin tomar en cuenta primero la salida.

7.4.1.1 Recorridos necesarios en el árbol para alcance de definiciones.

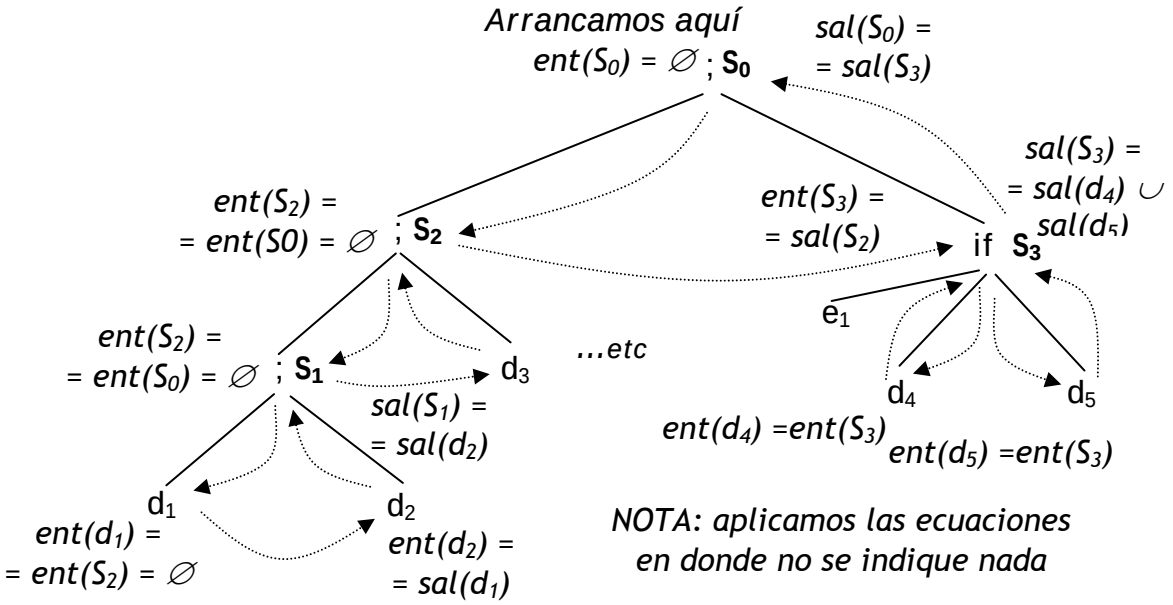
Veamos como obtener los atributos en el árbol sintáctico y su cálculo mediante recorridos en el árbol mediante un ejemplo:

1º. Calculamos genera y desactiva de forma sintetizada con un primer recorrido ascendente de árbol sintáctico.



2º. Partimos de que $ent[S_0] = \varnothing$, siendo S_0 el programa completo (a la entrada del programa completo no hay ninguna definición. A partir de ahí recorremos el árbol en profundidad de izquierda a derecha y vamos obteniendo la entrada

de forma heredada, y calculando la salida de forma sintetizada escalando en el árbol.



Como ya se ha indicado, es muy importante darse cuenta de que, en el caso de los lazos (do), esta secuencia no es aplicable porque ent[S_1] = ent[S] ∪ sal[S_1], es decir, necesitamos la salida para conseguir la entrada. Luego veremos en este caso como se soluciona con una definición equivalente.

Para resolver el problema con los lazos, existe una definición equivalente para ent[S_1] que nos va a servir para realizar los cálculos normalmente (primero calculamos de forma sintetizada genera y desactiva y luego, heredando la entrada, calculamos la salida), que es:

ent[S_1] = ent[S] ∪ gen[S_1]

Vamos a demostrarlo. Partimos de que:

ent[S_1] = ent[S] ∪ sal[S_1]
sal[S_1] = gen[S_1] ∪ (ent[S_1] – desact[S_1])

vamos a escribirlo como:

E1 = E ∪ X1
X1 = G1 ∪ (E1 – D1)

E1 y X1 son variables, las otras tres son constantes.

Al comenzar la primera iteración, X1 = 0 y como entrada de la primera iteración tenemos:

$E1^1$=E

La salida de la primera iteración es:
$X1^1 = G1 \cup (E1^1 - D1) = G1 \cup (E - D1)$

La entrada de la segunda iteración es:
$E1^2 = E \cup X1^1 = E \cup G1 \cup (E - D1) = E \cup G1$

La salida de la segunda iteración es:
$X1^2 = G1 \cup (E1^2 - D1) = G1 \cup (E \cup G1 - D1) = G1 \cup (E - D1)$

Si continuamos con las iteraciones veremos que no hay variación alguna. Por lo tanto, hemos demostrado que:

$ent[S_1] = ent[S] \cup gen[S_1]$

7.4.2 Notación vectorial para representar genera y desactiva.

La notación que utilizaremos para los conjuntos **genera** y **desactiva** será en formato vectorial:

$V = (a_1, a_2, \ldots, a_n)$

n: número de definiciones
Si $a_i = 0$ significa que se ha generado o desactivado (según sea uno u otro vector) una definición, si su valor es 1 es todo lo contrario.

Con esta representación, los cálculos se realizarán de la siguiente forma:

A – B se calcula como	VA /\ ¬ VB	(AND lógico con VB negado)
A ∪ B se calcula como	VA \/ VB	(OR lógico entre VA y VB)
A ∩ B se calcula como	VA /\ VB	(AND lógico entre VA y VB)

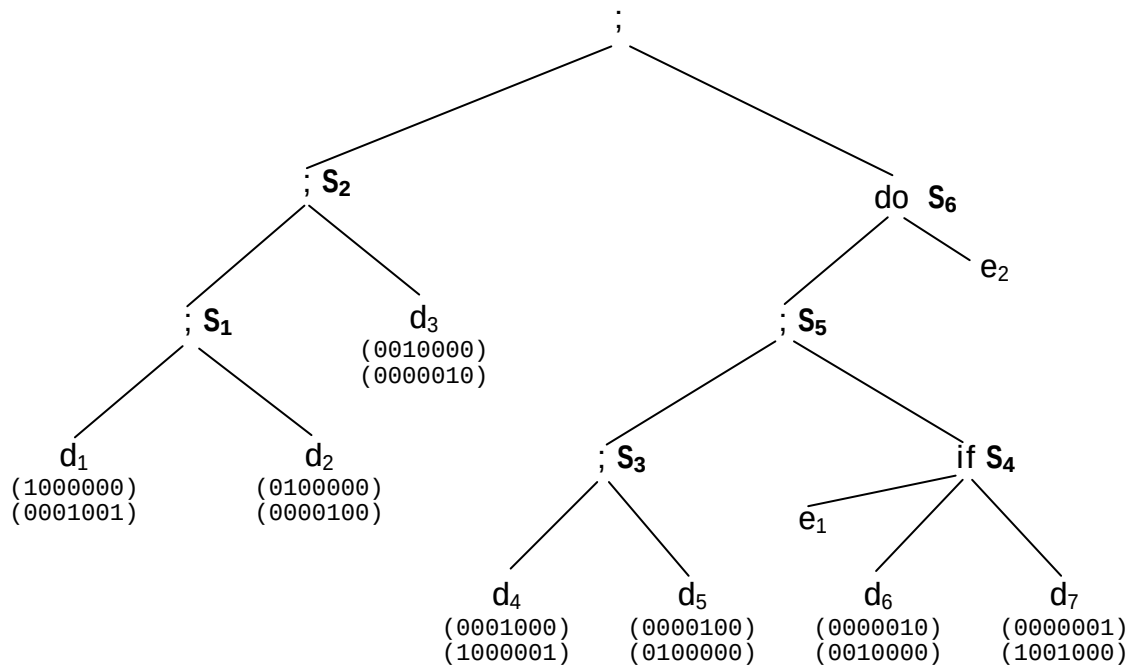
Ejemplo.-

```
/* d1 */ i := m-1;
/* d2 */ j := n;
/* d3 */ a := u1;
do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e1 then
/* d6 */               a := u2;
               else
/* d7 */               i := u3;
while e2
```

NOTA: con $d_1, d_2, \ldots, d_n$ indicamos las definiciones que se realiza en esa proposición (ese será el orden utilizado en los vectores **genera** y **desactiva**).

También es necesario hacer notar que ambos conjuntos son atributos sintetizados, que se obtienen de forma ascendente.

Veamos el árbol sintáctico, con las definiciones en las hojas, y los vectores genera y desactiva:



$gen[S_1] = gen[d_2] \cup (gen[d_1] - desact[d_2]) =$
$= (0100000) \cup ((1000000) - (0000100)) = (1100000)$

$desact[S_1] = desact\ [d_2] \cup (desact[d_1] - gen[d_2]) =$
$= (0000100) \cup ((0001001) - (0100000)) = (0001101)$

$gen[S_2] = gen[d_3] \cup (gen[S_1] - desact[d_3]) =$
$= (0010000) \cup ((1100000) - (0000010)) = (1110000)$

$desact[S_2] = desact\ [d_3] \cup (desact[S_1] - gen[d_3]) =$
$= (0000010) \cup ((0001101) - (0010000)) = (0001111)$

$gen[S_3] = gen[d_5] \cup (gen[d_4] - desact[d_5]) =$
$= (0000100) \cup ((0001000) - (0100000)) = (0001100)$

$desact[S_3] = desact[d_5] \cup (desact[d_4] - gen[d_5]) =$
$= (0100000) \cup ((1000001) - (0000100)) = (1100001)$

$gen\ [S_4] = gen[d_6] \cup gen[d_7] = (0000010) \cup (0000001) = (0000011)$
$desact[S_4] = desact[d_6] \cap desact[d_7] = (0000000)$

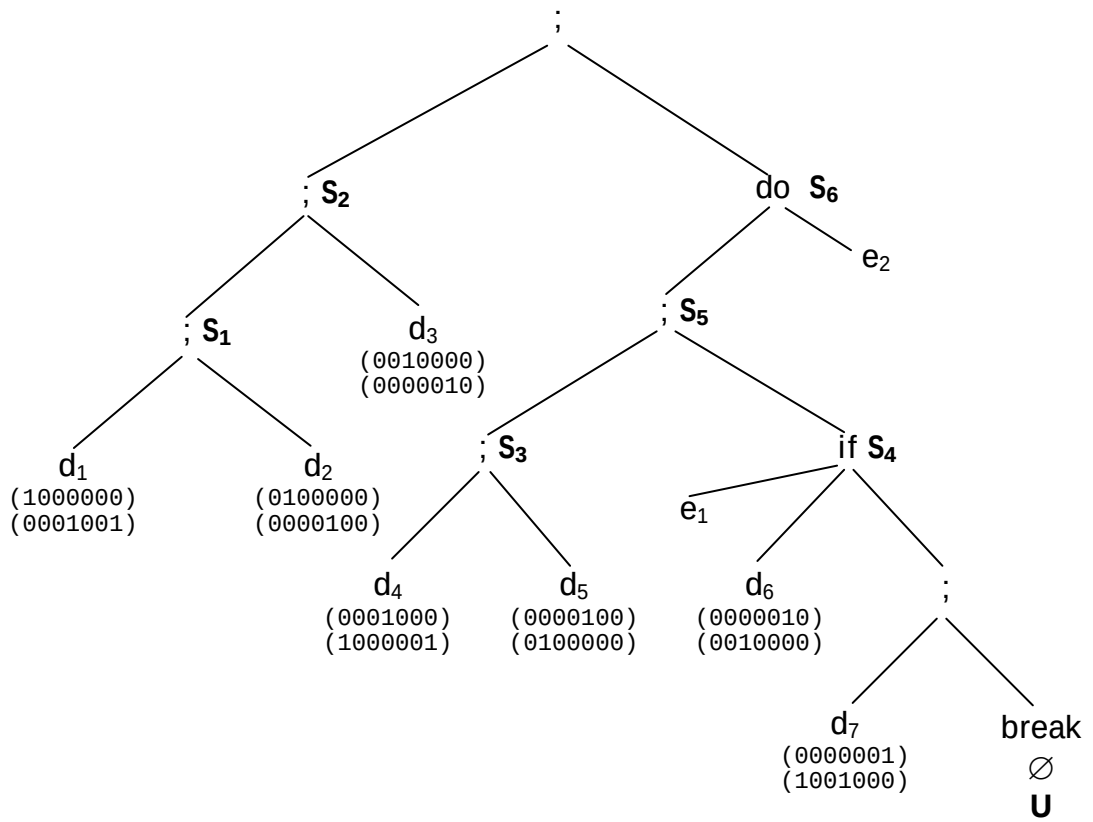
$gen[S_5] = gen[S_4] \cup (gen[S_3] - desact[S_4]) =$
$= (0000011) \cup ((0001100) - (0000000)) = (0001111)$

desact(S_5) = desact [S_4] $\cup$ (desact[S_3] – gen[S_4]) =
= (0000000) $\cup$ ((1100001) – (0000011)) = (1100000)

gen[S_6] = gen[S_5] = (0001111)
desact[S_6] = desact[S_5] = (1100000)

Vamos a realizar una modificación en nuestro código, introduciendo una instrucción **break** dentro del lazo do-while (salta al final del lazo), de la siguiente forma:

```
/* d1 */ i := m-1;
/* d2 */ j := n;
/* d3 */ a := u1;
do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e1 then
/* d6 */               a := u2;
               else begin
/* d7 */               i := u3;
                       break
               end
while e2
```



Para realizar los cálculos, donde esta la instrucción **break** decimos que no se genera nada (∅), en nuestro ejemplo (0000000) y se desactiva todo (**U**), en nuestro ejemplo (1111111) y realizamos los cálculos normalmente. Esta es una

postura también conservadora porque no se puede llegar al final de una secuencia de proposiciones que finalice con una proposición break.

7.5 Solución iterativa de las ecuaciones de flujo de datos.

7.5.1 Análisis de alcance de definiciones.

Vamos a definir bloques (B) básicos, considerando cada bloque como una proposición que es cascada de una o varias proposiciones de asignación. También se definen sal[B], gen[B], desact[B] y ent[B] de la misma forma que en los apartados anteriores.

Algoritmo iterativo para alcance de definiciones:

Partimos de que se ha calculado gen[B] y desact[B] para cada bloque B. Y además partimos de que:

$ent[B] = \cup_P sal[P]$ La unión de los P, bloques predecesores de B.
$sal[B] = gen[B] \cup (ent[B] - desact[B])$

Algoritmo:

Entrada: genera y desactiva para cada B_i del grafo.
Salida: ent[B_i], sal[B_i]

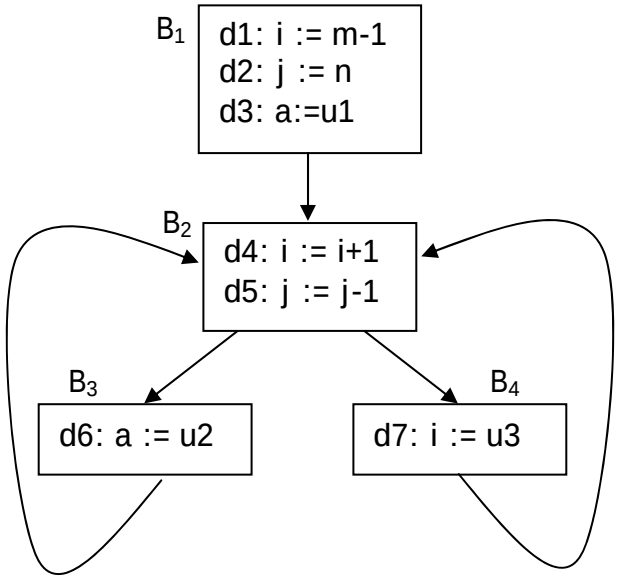
```
ent[Bi] = ∅
for cada Bi do sal[Bi] = gen[Bi];
cambio := true;
while cambio do begin
        cambio := false;
        for cada bloque B do begin
                ent[Bi] := ∪P sal[P];
                salant := sal[Bi];
                sal[Bi] := gen[Bi] ∪ (ent[Bi] – desact[Bi]);
                if sal[Bi] ≠ salant then cambio:= true
        end
end
```

Ejemplo.-

```
/* d1 */ i := m-1;
/* d2 */ j := n;
/* d3 */ a := u1;
do
/* d4 */       i := i+1;
/* d5 */       j := j-1;
               if e1 then
```

```
/* d6 */            a := u2;
          else
/* d7 */            i := u3;
while e2
```

Vamos a ver el grafo de flujo:



$gen[B_1] = \{d_1, d_2, d_3\}$
$desact[B_1] = \{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4, d_5\}$
$desact[B_2] = \{d_1, d_2, d_7\}$

$gen[B_3] = \{d_6\}$
$desact[B_3] = \{d_3\}$

$gen[B_4] = \{d_7\}$
$desact[B_4] = \{d_1, d_4\}$

En formato vectorial, los conjuntos genera y desactiva son:

BLOQUE	**genera**	**desactiva**
B1	1110000	0001111
B2	0001100	1100001
B3	0000010	0010000
B4	0000001	1001000

Vamos a ver como calcular las entradas en los distintos bloques según sus predecesores:

$ent[B_1] = \varnothing$
$ent[B_2] = sal[B_1] \cup sal[B_3] \cup sal[B_4]$
$ent[B_3] = sal[B_2]$
$ent[B_4] = sal[B_2]$

Si aplicamos el algoritmo:

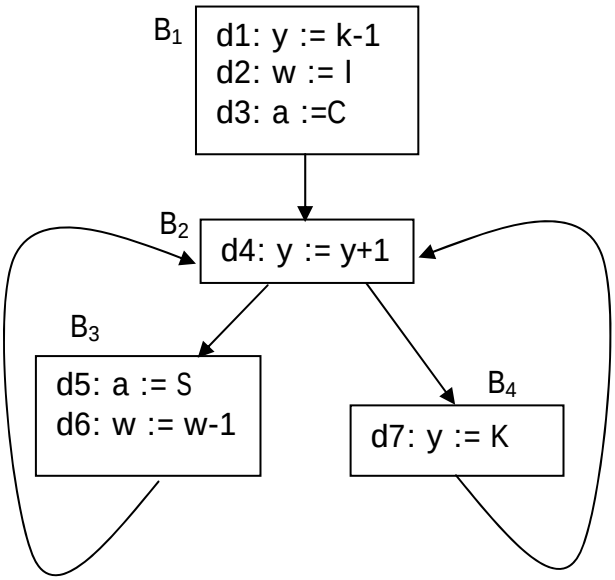
	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**ent**	**sal**	**ent**	**sal**	**ent**	**sal**
B1	∅	1110000	∅	*1110000*	∅	*1110000*
B2	∅	0001100	1110011	*0011110*	1111111	*0011110*
B3	∅	0000010	*0011110*	*0001110*	0011110	*0001110*
B4	∅	0000001	*0011110*	*0010111*	0011110	*0010111*

Las casillas en cursiva y negrita, conjuntos salida, tienen el mismo valor, con lo cual paramos el algoritmo y ya tenemos la entrada y la salida para cada uno de los bloques. NOTA: La entrada de la Iteración 1 en B3 y B4 (cursiva)se puede tomar enrelación a la salida de B2 de esa misma iteración o de la anterior. No afecta al resultado, sólo al número de iteraciones.

Ejemplo.-

```
/* d1 */    y := k-1;
/* d2 */    w := l;
/* d3 */    a := C;
            do
/* d4 */            y := y+1;
                    if condicion1 then
/* d5 */                                a := S;
/* d6 */                                w := w-1;
                    else
/* d7 */                        y := K;
            while condicion2
```

El grafo de flujo sería el siguiente:



$gen[B_1] = \{d_1, d_2, d_3\}$
$desact[B_1] = \{d_4, d_5, d_6, d_7\}$

$gen[B_2] = \{d_4\}$
$desact[B_2] = \{d_1, d_7\}$

$gen[B_3] = \{d_5, d_6\}$
$desact[B_3] = \{d_3, d_2\}$

$gen[B_4] = \{d_7\}$
$desact[B_4] = \{d_1, d_4\}$

En formato vectorial, los conjuntos genera y desactiva son:

BLOQUE	genera	desactiva
B1	1110000	0001111
B2	0001000	1000001
B3	0000110	0110000
B4	0000001	1001000

Vamos a ver como calcular las entradas en los distintos bloques según sus predecesores:

ent[B_1] = ∅
ent[B_2] = sal[B_1] ∪ sal[B_3] ∪ sal[B_4]
ent[B_3] = sal[B_2]
ent[B_4] = sal[B_2]

Si aplicamos el algoritmo:

	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**ent**	**Sal**	**Ent**	**sal**	**ent**	**sal**
B1	∅	1110000	∅	***1110000***	∅	***1110000***
B2	∅	0001000	1110111	***0111110***	1111111	***0111110***
B3	∅	0000110	0111110	***0001110***	0111110	***0001110***
B4	∅	0000001	0111110	***0110111***	0111110	***0110111***

Las casillas en cursiva y negrita, conjuntos salida, tienen el mismo valor, con lo cual paramos el algoritmo y ya tenemos la entrada y la salida para cada uno de los bloques.

7.5.2 Análisis de expresiones disponibles.

Una expresión x+y está disponible en un punto p si todo camino, no necesariamente sin lazos, desde el nodo inicial hasta p evalua x+y y después de la última evaluación antes de p no hay asignaciones posteriores de x ni de y.

Un bloque B_i desactiva una expresión x+y si asigna un valor a x o a y.

Un bloque B_i genera una expresión x+y si evalua x+y y no genera una redefinición posterior de x o de y.

e_gen[B_i] es el conjunto de las expresiones generadas en el bloque B_i.
e_desact[B_i] es el conjunto de las expresiones desactivadas en el bloque B_i.

U es el conjunto universal , todas las expresiones que hay en el programa.
∅ es el complementario de **U**.

Las ecuaciones a aplicar son:

sal[B_i] = e_gen[B_i] ∪ (ent[B_i] – e_desact[B_i])
ent[B_i] = $\cap_p$ sal[P_i] si Bi no es el bloque inicial
ent[B_1] = ∅ siendo B_1 el bloque inicial

Aquí utilizamos la ∩ de la salida de los predecesores porque para que una expresión esté disponible en un bloque tiene que salir de todos sus predecesores (una vez más, utilizamos una estrategia conservadora).

Ejemplo.-

(1) a = b + c

-1- Disponible la expresión 1

(2) b = a + d

-2- Disponible la expresión 2 (se ha cambiado b y la 1 ya pasa a no estar disponible)

(3) c = b + c

-3- Disponible la expresión 2 (la 3 ya no es disponible inmediatamente pues hemos cambiado el valor de c)

(4) d = a – d

-4- Ninguna es disponible (se ha cambiado el valor de d y la dos ya no está disponible y la propia expresión 4 ya cambia inmediatamente el valor de d).

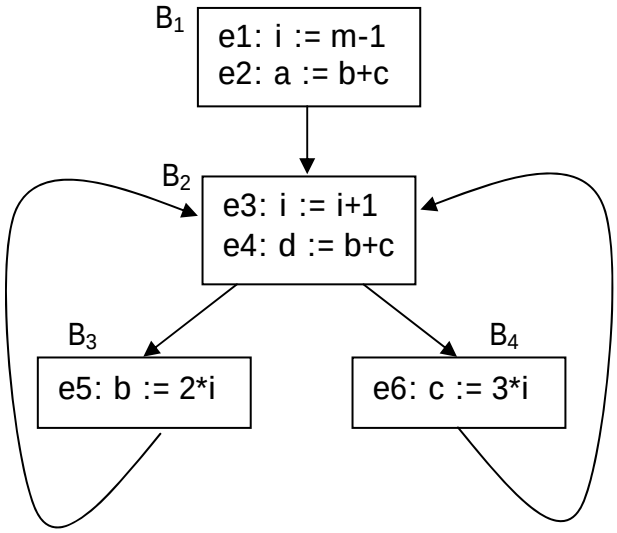
Algoritmo iterativo para expresiones disponibles:

ENTRADA: Grafo de flujo G, e_genera[B_i], e_desactiva[B_i]
ent[B_i] = $\varnothing$; /* B_1 es el bloque inicial */
sal[B_1] = e_gen[B_1];
for B $\neq B_1$ do sal[B] = **U** – e_desact[B];
cambio = true;

```
while cambio do begin
        cambio = false
        for B ≠ B1 do begin
                ent[B] = ∩p sal[P];
                salant = sal[B];
                sal[B] = e_gen[B] ∪ (ent[B] – e_desact[B]);
                if salant ≠ sal[B] then cambio = true;
        end;
end;
```

Ejemplo.-

```
/* e1 */            i := m-1;
/* e2 */            a := b+c;
                do
/* e3 */            i := i+1;
/* e4 */            d := b+c;
                i   f e1 then
/* e5 */                    b := 2*i;
                    else
/* e6 */                    c := 3*i;
                while e2
```



$e\_gen[B_1] = \{e_1, e_2\}$
$e\_desact[B_1] = \{e_3, e_5, e_6\}$

$e\_gen[B_2] = \{e_4\}$
$e\_desact[B_2] = \{e_3, e_5, e_6\}$

$e\_gen[B_3] = \{e_5\}$
$e\_desact[B_3] = \{e_2, e_4\}$

$e\_gen[B_4] = \{e_6\}$
$e\_desact[B_4] = \{e_2, e_4\}$

En formato vectorial, e_genera y e_desactiva son:

BLOQUE	**e_genera**	**e_desactiva**
B1	110000	001011
B2	000100	001011
B3	000010	010100
B4	000001	010100

Vamos a ver como calcular las entradas en los distintos bloques según sus predecesores:

$ent[B_1] = \varnothing$
$ent[B_2] = sal[B_1] \cap sal[B_3] \cap sal[B_4]$
$ent[B_3] = sal[B_2]$
$ent[B_4] = sal[B_2]$

Para las salidas aplicamos:

$sal[B_i] = e\_gen[B_i] \cup (ent[B_i] - e\_desact[B_i])$

	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**ent**	**Sal**	**Ent**	**sal**	**ent**	**sal**
B1	∅	110000	∅	*110000*	∅	*110000*
B2	∅	110100	100000	*100100*	100000	*100100*
B3	∅	101011	110100	*100010*	100100	*100010*
B4	∅	101011	110100	*100001*	100100	*100001*

En las dos últimas iteraciones vemos que **sal** se mantiene, entonces paramos el algoritmo y ya tenemos las expresiones que entran y salen en cada bloque.

7.5.3 Análisis de variables activas.

Una variable a es activa en un punto p se utiliza el valor a en algún camino que comience en p, en caso contrario se dirá que a está desactiva.

Como se puede deducir, el análisis en este caso se realiza en dirección opuesta al flujo de ejecución del programa. Como sabemos que una variable está activa en un punto del programa si se usa en un punto posterior, esa información la recogeremos analizando el programa en este sentido.

Entrada es el conjunto de variables activas al comienzo del bloque.

Salida es el conjunto de variables activas a la salida del bloque.

Definidas son el conjunto de variables a las que se les ha asignado definitivamente un valor en en bloque.

Uso son el conjunto de variables cuyos valores se utilizan antes de cualquier definición de la variable.

Las ecuaciones a aplicar son las siguientes:

$ent[B_i] = uso[B_i] \cup (sal[B_i] - def[B_i])$
$sal[B_i] = \cup_S ent[S_i]$

La primera ecuación nos indica que una variable está activa al entrar en un bloque si se utiliza antes de una redefinición en el bloque o si está activa al salir del bloque y dentro no se redefine.

La segunda ecuación nos indica que una variable está activa al salir de un bloque si, y sólo si, está activa al entrar en uno de sus sucesores.

Algoritmo iterativo de análisis de variables activas:

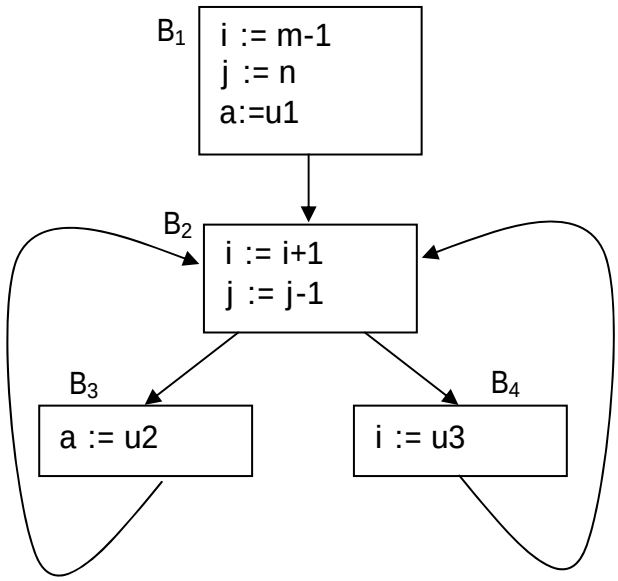
Entrada: Un grafo de flujo G, $def[B_i]$ y $uso[B_i]$
Salida: sal[Bi], es decir, el conjunto de variables activas a la salida de cada bloque B del grafo de flujo.

```
for cada bloque B do sal[B] := ∅;
for cada bloque B do ent[B] := uso[B];
while ocurren cambios en los conjuntos ent do
        for cada bloque B do begin
                sal[B] := ∪S ent[S];
                ent[B] := uso[B] ∪ (sal[B] – def[B])
        end
end
```

Ejemplo.-

```
i := m-1;
j := n;
a := u1;
do
        i := i+1;
        j := j-1;
                if e1 then
                        a := u2;
                else
                        i := u3;
while e2
```

Vamos a ver el grafo de flujo:



$def[B_1] = \{i, j, a\}$
$uso[B_1] = \{m, n, u1\}$

$def[B_2] = \{i, j\}$
$uso[B_2] = \{i, j\}$

$def[B_3] = \{a\}$
$uso[B_3] = \{u2\}$

$def[B_4] = \{i\}$
$uso[B_4] = \{u3\}$

Para la notación vectorial, vamos a utilizar el orden de aparición de las variables en el programa, esto es:

(i, m, j, n, a, u1, u2, u3)

En formato vectorial, los conjuntos definición y uso son:

BLOQUE	definición	uso
B1	10101000	01010100
B2	10100000	10100000
B3	00001000	00000010
B4	10000000	00000001

Vamos a ver como calcular las salidas en los distintos bloques según sus sucesores:

$sal[B_1] = ent[B_2]$
$sal[B_2] = ent[B_3] \cup ent[B_4]$
$sal[B_3] = ent[B_2]$
$sal[B_4] = ent[B_2]$

Si aplicamos el algoritmo:

	Inicialmente		*Iteración 1*		*Iteración 2*	
BLOQUE	**sal**	**Ent**	**sal**	**ent**	**sal**	**ent**
B1	∅	01010100	10100000	01010100	10100011	***01010111***
B2	∅	10100000	00000011	10100011	10100011	***10100011***
B3	∅	00000010	10100000	10100010	10100011	***10100011***
B4	∅	00000001	10100000	00100001	10100011	***00100011***

	Iteración 3	
BLOQUE	**sal**	**ent**
B1	10100011	***01010111***
B2	10100011	***10100011***
B3	10100011	***10100011***
B4	10100011	***00100011***

Las casillas en cursiva y negrita, conjuntos entrada, tienen el mismo valor, con lo cual paramos el algoritmo y ya tenemos la entrada y la salida para cada uno de los bloques.

8 ERRORES.

En este capítulo trataremos de los errores que pueden aparecer durante las tareas de compilación principalmente, basándonos en criterios ortográficos, sintácticos y semánticos.

8.1 Tipos de errores.

Los distintos tipos de errores que podemos encontrarnos en un programa son los siguientes:

1) **Errores léxicos,** asociados a las tareas de análisis léxico. Algunos podrían ser la detección de un carácter que no pertenece al vocabulario, escritura incorrecta de un identificador, aparición de más de un punto decimal en una constante real, etc.

   Por ejemplo.- Cuando en el análisis léxico realizábamos una implementación mediante un autómata, las casillas en blanco de la tabla de transiciones significaban errores. Estos errores es preciso identificarlos recorriendo el autómata, por ejemplo tenemos errores como la aparición de dos signos en un número, existencia de más de un punto decimal, etc.

2) **Errores sintácticos**, se producen cuando el analizador sintáctico o parser no es capaz de obtener el árbol de derivación para la tira de entrada.

   Por ejemplo.- La tira "A = 3 * I + (J * R" tiene un error, pues faltan los paréntesis.

3) **Errores semánticos**, se producen cuando utilizamos variables no declaradas, existen declaraciones múltiples, inconsistencia en parámetros, etc.

4) **Errores de compilación**, son los que genera el compilador cuando se exceden límites marcados por él mismo. Algunos son:
   - Limitaciones en memoria lo que implica limitaciones en el tamaño de la tabla de símbolos, número FOR anidados, etc.
   - Limitaciones en las pilas de análisis sintácticos y generación de código, condicionando de esta forma la complejidad del código.
   - Limitaciones en el número de bloques que se pueden definir en el programa.

   Todos los límites del compilador deberían estar claramente definidos.

5) **Errores de ejecución**, son los que se producen durante la ejecución del código objeto generado. Algunos son:
   - Aritméticos, como por ejemplo una división por 0 o una raíz cuadrada de un número negativo.
   - Acceso a matrices fuera de rango.

- Acceso a ficheros no abiertos.
- Problemas en punteros a ficheros (por ejemplo intentar acceder a un dato más allá del fin de fichero).
- Problemas de memoria en tiempo de ejecución al intentar reservar memoria dinámicamente cuando no hay disponibilidad.
- Limitaciones de la pila de ejecución (una de las situaciones más típicas es el abuso en la recursividad de partes del código).

8.2 Recuperación de errores léxico-gráficos.

En este apartado vamos a centrarnos en la corrección automática de errores léxico-gráficos, producidos por una escritura incorrecta de nombres de palabras reservadas, variables, etc. Veremos errores de intercambio de caracteres, omisión o borrado de caracteres y de inclusión de caracteres.

8.2.1 Corrección de errores de sustitución.

Este tipo de errores se producen cuando se ha introducido en algún elemento léxico cambios en ciertos caracteres.

Vamos a definir el **operador de cambio C**.

$C(a) = T - \{a\}$

$x \in T^*$

$C^1(x) = \{ y \ / \ y \in T^* \wedge$ y resultado de 1 cambio en x$\}$
$C^2(x) = \{ y \ / \ y \in T^* \wedge$ y resultado de 2 cambios en x$\}$
…
$C^n(x) = \{ y \ / \ y \in T^* \wedge$ y resultado de n cambios en x$\}$
Nota: Si $|x| < n$, entonces $C^n(x) = \varnothing$

Notación: Si tenemos que: $A \rightarrow \alpha_1 \mid \alpha_2 1 \ldots \mid \alpha_m$
Entonces escribimos: $A = \alpha_1 + \alpha_2 + \ldots + \alpha_m$

Si tenemos una G = (N, T, P, S), crearemos una G´=(N, T, P´, S), en la que las reglas serán: P´ = P $\cup$ {reglas de los operadores de cambio}. Por lo tanto P´:

$A \rightarrow \alpha_1$	$A \rightarrow C^1(\alpha_1)$	$A \rightarrow C^2(\alpha_1)$	…	$A \rightarrow C^n(\alpha_1)$
$A \rightarrow \alpha_2$	$A \rightarrow C^1(\alpha_2)$	$A \rightarrow C^2(\alpha_2)$	…	$A \rightarrow C^n(\alpha_2)$
…	…	…	…	…
$A \rightarrow \alpha_m$	$A \rightarrow C^1(\alpha_m)$	$A \rightarrow C^2(\alpha_m)$	…	$A \rightarrow C^n(\alpha_m)$

De esta forma tenemos una gramática G´ que considera las gramáticas bien escritas y las gramáticas mal escritas con errores de sustitución. Esto es:

$A = \alpha_1 + \alpha_2 + \ldots + \alpha_m + C^1(\alpha_1) + C^1(\alpha_2) + \ldots + C^1(\alpha_M) + \ldots + C^2(\alpha_1) + C^2(\alpha_2) + \ldots + C^2(\alpha_m) + \ldots + C^n(\alpha_1) + C^n(\alpha_2) + \ldots + C^n(\alpha_m)$
El nuevo lenguaje será:

$L(G') = \{ y \,/\, y \in T^* \wedge y \in C^n(x) \wedge x \in L(G), n \geq 0\}$

Ejemplo.- Sea la gramática:

$G = (\{S, B\}, \{a, b\}, P, S\}$

P: $S \rightarrow aB \mid aSa$
$B \rightarrow b$

$L(G) = \{a^n aba^n \,/\, n \geq 0\}$

Vamos a generar una gramática $G' = (N, T, P', S)$. Para ello comenzaremos por calcular los cambios en las partes derechas de las reglas:

$C^1(aB) = C^1(a)B = bB$
$C^1(aSa) = C^1(a)Ss + aSC^1(a) = bSa + aSb$
$C^1(b) = a$
$C^2(aSa) = C^1(a)SC^1(a) = bSb$

Por lo tanto las nuevas reglas (P´) son:

$S \rightarrow aB \mid aSa \mid bB \mid bSa \mid aSb \mid bSb$
$B \rightarrow b \mid a$

Ahora generamos un esquema de traducción (EDT) para realizar la corrección:

EDT (N, Te, Ts, P, S), cuyas reglas P son de la siguiente forma:

$A \rightarrow \alpha_i, \alpha_i$ cuando no existen errores
$A \rightarrow \mu, \alpha_i$ cuando $\mu \in C^1(\alpha_i)$
$A \rightarrow \beta, \alpha_i$ cuando $\beta \in C^2(\alpha_i)$
… …
$A \rightarrow \delta, \alpha_i$ cuando $\delta \in C^n(\alpha_i)$

Las reglas P de nuestro EDT son:

$S \rightarrow aB, aB$
$S \rightarrow aSa, aSa$
$B \rightarrow b, b$ Estas tres reglas cuando no hay errores.

$S \rightarrow bB, aB$
$S \rightarrow bSa, aSa$
$S \rightarrow aSb, aSa$ Estas tres reglas cuando hay un error

$S \rightarrow bSb, aSa$ Cuando hay dos errores.

8.2.2 Corrección de errores de borrado.

Este tipo de errores se producen cuando se ha omitido algún carácter en un componente léxico.

Vamos a definir el **operador de borrado B**.

$B(a) = \lambda$ (ó ε), $\forall\ a \in T$

$Bn(x) =$ $\{\varnothing$ si $|x| < n\}$
$\{y\ /\ y \in T^* \wedge$ resultado de eliminar n caracteres en x$\}$

Ejemplo.-

$G=(\{S, B\}, \{a, b\}, P, S)$

P: $S \rightarrow aB$
$S \rightarrow aSa$
$B \rightarrow b$

$G' = (N, T, P', S)$

P´: $B^1(aB) = B$
$B^1(aSa) = B^1(a)Sa + aSB^1(a) = Sa + aS$
$B^1(b) = \lambda$
$B^2(aSa) = B^1(a)SB^1(a) = S$

Al igual que en el caso de cambio, también utilizamos un EDT, de la siguiente forma:

P:	$S \rightarrow aB, aB$	
	$S \rightarrow aSa, aSa$	
	$B \rightarrow b, b$	Sin errores
	$S \rightarrow B, aB$	
	$S \rightarrow Sa, aSa$	
	$S \rightarrow aS, aSa$	
	$B \rightarrow \lambda, b$	Un error
	$S \rightarrow S, aSa$	Dos errores

NOTA: Para realizar correciones ocurre lo mismo que en los códigos Hamming, según la distancia podremos detectar, detectar y corregir, etc.

8.2.3 Corrección de errores de inclusión.

Estos errores aparecen al introducir caracteres no deseados dentro de los elementos léxicos del lenguaje.

Vamos a definir el **operador de inclusión I**.

$I(a) = Ta + aT \qquad \forall\ a \in T$

$T = \{a_1, a_2, \ldots, a_n\}$

$I(a) = aa_1 + aa_2 + \ldots + aa_n + a_1a + a_2a + \ldots a_na$

La gramática G´también la construimos con un EDT = (N, Te, Ts, R, S)

G=({S, B}, {a, b}, P, S)

P: $S \rightarrow aB$
$S \rightarrow aSa$
$B \rightarrow b$

G´= (N, T, P´, S)

P´: $S \rightarrow aaB, aB$
$S \rightarrow baB, aB$
$B \rightarrow bb, b$
$B \rightarrow ab, b$
$B \rightarrow ba, b$
…

8.3 Análisis sintáctico en "modo pánico".

Al realizar el árbol sintáctico, cuando no se puede realizar el análisis, el compilador busca un punto después del error a partir del cual se puede seguir realizando el análisis sintáctico. Sobre el trozo de código que se ha "saltado" no muestra todos los errores.

Típicamente se utilizan las estructuras BEGIN-END para saltar bloques en los que se encuentran errores.

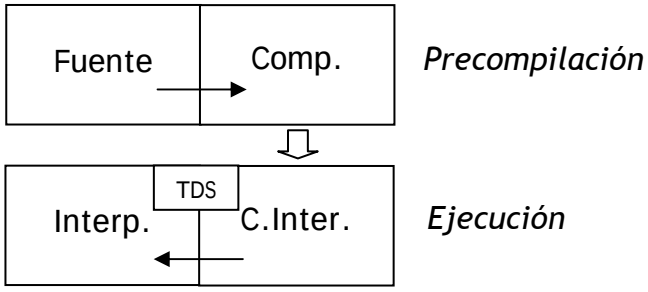
9 INTÉRPRETES.

Los intérpretes, en la primera generación de ordenadores, han tenido mucho éxito por los problemas de memoria existentes. Muchas veces el programa fuente y el intérprete ocupaban memos memoria que el programa fuente y el compilador e incluso que el objeto.

Hoy en día han caído en desuso debido precisamente a la misma causa, hoy no tenemos problemas con la memoria de los ordenadores.

Además tienen el problema de la detección de errores. Puede llegarse a un punto del programa en que exista un error y todo el trabajo anterior no sirve para nada.

Por ello, se optó por una opción intermedia, se realiza una compilación previa a un lenguaje intermedio, que además servirá para detectar errores. Luego el intérprete actúa sobre este lenguaje intermedio. El esquema es el siguiente:



Es decir, tenemos dos tipos de intérprete:

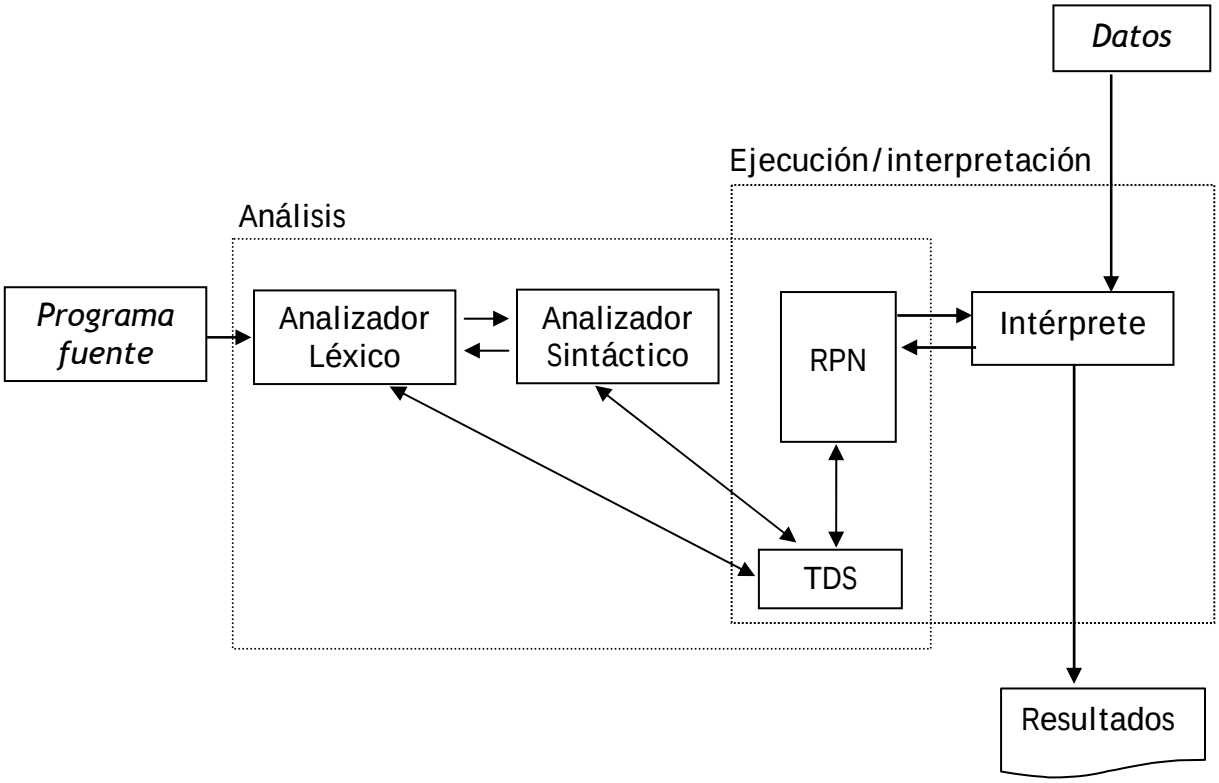
a) Intérprete puro.- En el que no hay precompilación.
b) Inérprete actual.- Tenemos un código intermedio que hará el reconocimiento léxico, sintáctico, etc. Posteriormente el módulo de interpretación coge línea a línea el código intermedio y lo va ejecutando.

9.1 Estructura de un intérprete actual.

En un intérprete actual tendremos una primera fase de análisis que será similar a la que nos encontramos en los compiladores.

Posteriormente se realiza la fase de interpretación y ejecución, que utilizará la TDS generada anteriormente y en la que hemos supuesto la utilización de un lenguaje intermedio en RPN.

Veamos el siguiente esquema:



Ejemplo.-

Un programa en BASIC:

```
A=9 ;
B=8 ;
B=B+A ;
WRITE B ;
GOTO 1000
```

Se traduce a código intermedio en RPN, en forma compacta, de la siguiente forma:

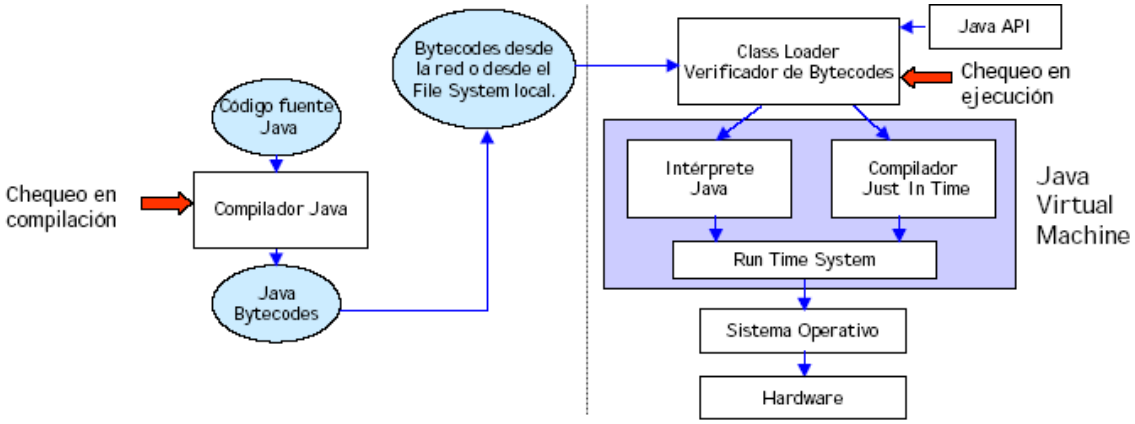
A 9 := ; ; B 8 := ; B B A + := ; B WRITE ; 9 GOTO ;

El intérprete lo ejecutaría utilizando la siguiente programación (PCP es el puntero a la cabeza de la pila):

```
case V[pos] of
        ident, entero: apilar(V[pos]); p := p+1;
        "+": sumar los elementos PCP y PCP-1 y sustituir en la pila;
        "write": representar en pantalla el PCP;
        ":=": evaluar elemento PCP y ponerlo en la dirección del de PCP-1;
end
```

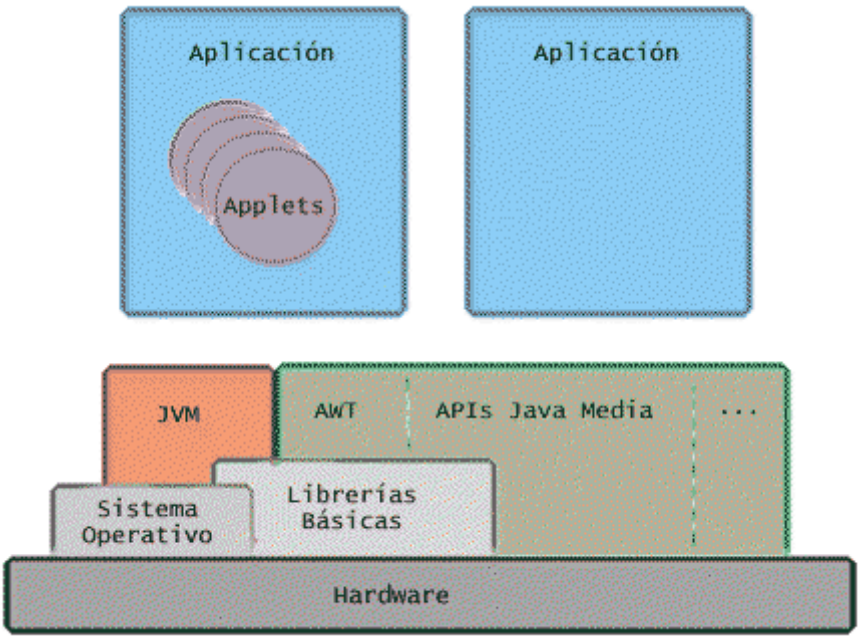
9.2 Arquitectura “neutral” de Java

El compilador Java compila su código a un fichero objeto de formato independiente de la arquitectura de la máquina en que se ejecutará. Cualquier máquina que tenga el sistema de ejecución (run-time) puede ejecutar ese código objeto, sin importar en modo alguno la máquina en que ha sido generado. Actualmente existen sistemas run-time para Solaris 2.x, SunOs 4.1.x, Windows 95, Windows NT, Linux, Irix, Aix, Mac, etc.



El código fuente Java se "compila" a un código de bytes de alto nivel independiente de la máquina. Este código (byte-codes) está diseñado para ejecutarse en una máquina hipotética que es implementada por un sistema run-time, que sí es dependiente de la máquina.

En una representación en que tuviésemos que indicar todos los elementos que forman parte de la arquitectura de Java sobre una plataforma genérica, obtendríamos una figura como la siguiente:



En ella podemos ver que lo verdaderamente dependiente del sistema es la Máquina Virtual Java (JVM) y las librerías fundamentales, que también nos permitirían acceder directamente al hardware de la máquina. Además, habrá

APIs de Java que también entren en contacto directo con el hardware y serán dependientes de la máquina, como ejemplo de este tipo de APIs podemos citar:

- Java 2D: gráficos 2D y manipulación de imágenes.
- Java Media Framework : Elementos críticos en el tiempo: audio, video...
- Java 3D: Gráficos 3D y su manipulación.
- Etc.

La verdad es que Java para conseguir ser un lenguaje independiente del sistema operativo y del procesador que incorpore la máquina utilizada, es tanto interpretado como compilado. Y esto no es ningún contrasentido, me explico, el código fuente escrito con cualquier editor se compila generando el byte-code. Este código intermedio es de muy bajo nivel, pero sin alcanzar las instrucciones máquina propias de cada plataforma y no tiene nada que ver con el p-code de Visual Basic. El byte-code corresponde al 80% de las instrucciones de la aplicación. Ese mismo código es el que se puede ejecutar sobre cualquier plataforma. Para ello hace falta el run-time, que sí es completamente dependiente de la máquina y del sistema operativo, que interpreta dinámicamente el byte-code y añade el 20% de instrucciones que faltaban para su ejecución. Con este sistema es fácil crear aplicaciones multiplataforma, pero para ejecutarlas es necesario que exista el run-time correspondiente al sistema operativo utilizado.

Aunque en las versiones iniciales de Java el motor de ejecución consistía de un interprete de códigos de operación, actualmente se utiliza la tecnología de "generación de código justo en el momento" (*Just-in-Time code generation*), en dónde las instrucciones que implementan a los métodos, se convierten en código nativo que se ejecuta directamente en la máquina sobre la que se subyace (siendo de esta forma dependiente de la plataforma). El código nativo se genera únicamente la primera vez que se ejecuta el código de operación Java, por lo que se logra un aumento considerable en el rendimiento de los programas.