Universidad
Europea de Madrid

LAUREATE INTERNATIONAL UNIVERSITIES

# GENERACIÓN DE CÓDIGO INTERMEDIO

# CÓDIGO DE TRES DIRECCIONES

© Todos los derechos de propiedad intelectual de esta obra pertenecen en exclusiva a la Universidad Europea de Madrid, S.L.U. Queda terminantemente prohibida la reproducción, puesta a disposición del público y en general cualquier otra forma de explotación de toda o parte de la misma.

La utilización no autorizada de esta obra, así como los perjuicios ocasionados en los derechos de propiedad intelectual e industrial de la Universidad Europea de Madrid, S.L.U., darán lugar al ejercicio de las acciones que legalmente le correspondan y, en su caso, a las responsabilidades que de dicho ejercicio se deriven.

# Índice

Presentación 4

Tipos de código de tres direcciones 5

Cuartetos 7

Tercetos 9

Tercetos indirectos 10

DDS para expresiones 11

Evaluación de arrays I 12

- Para un array de una dimensión (vector) 12
- Para un array de dos dimensiones (matriz) 12

Evaluación de arrays II 14

Evaluación de arrays III 16

Resumen 17

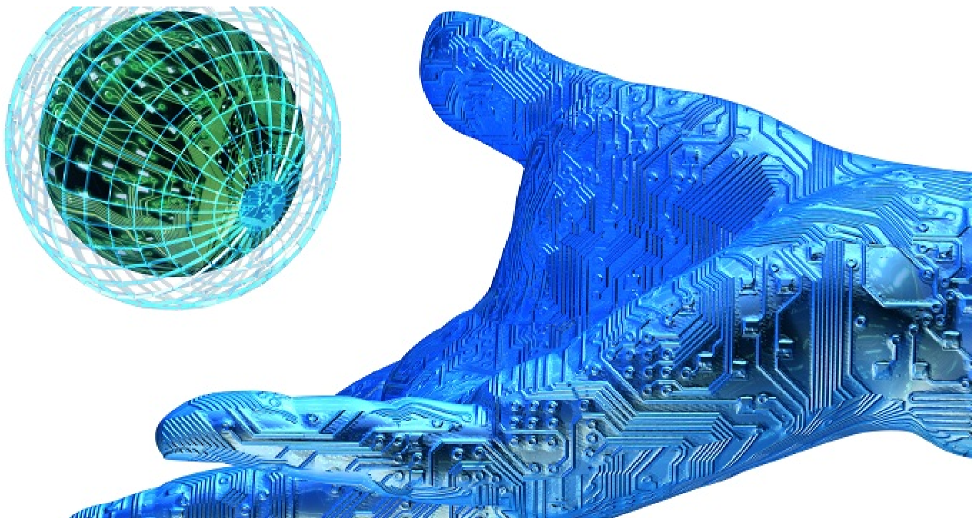


© Universidad Europea de Madrid. Todos los derechos reservados.

## Presentación

El objetivo de este tema es comprender las posibilidades y la casuística del código de tres direcciones. Para todo ello se alcanzarán los siguientes objetivos:

- Comprender qué es el código de tres direcciones y sus distintos tipos.
- Entender cómo se construyen los cuartetos, así como sus ventajas e inconvenientes.
- Entender cómo se construyen los tercetos, así como sus ventajas e inconvenientes.
- Entender cómo se construyen los tercetos indirectos, así como sus ventajas e inconvenientes.
- Entender la forma de generar código de tres direcciones para los *arrays* y conocer cómo se evalúan.





© Universidad Europea de Madrid. Todos los derechos reservados.

## Tipos de código de tres direcciones

Como ya sabemos tenemos dos tipos principales de generación de código intermedio, los gráficos representados por el **ASA y GDA y los lineales representados por el código de tres direcciones.** Como nos indica Louden (2004), la instrucción básica del código de tres direcciones está diseñada para representar la evaluación de expresiones aritméticas, aunque no de forma exclusiva, y tiene la siguiente forma general: x = y op z.

Los tipos de proposiciones de tres direcciones que nos podemos encontrar son (Aho et al, 1986 y 2008):

| | |
|---|---|
| Instrucciones de asignación binarias y unarias | a:= b op c, sería binaría, y a:= -b sería unaria. |
| Instrucciones de copia | a:= b. |
| Instrucciones de copia indexadas | a:= T[b] y viceversa T[b]:= a. |
| Saltos incondicionales | goto B1, donde B1 es una etiqueta simbólica que necesita ser traducida a una dirección de memoria. |
| Saltos condicionales en sus distintas modalidades | if (a) goto B1 o por ejemplo, if (a < b or c > d) goto B2. Hay que tener en cuenta que los bucles tipo while y for, se convierte en saltos condicionales. |
| Llamadas a procedimientos y sus retornos | call f(a, b,...), o bien x:= call f(a, b,...). |
| Asignaciones | De direcciones y punteros. |

El código de tres direcciones utiliza variables temporales, por lo que es necesario implementar una función que las vaya proporcionando.

**x, y o z**

Donde x, y o z (dos operandos y resultado, de ahí el nombre de tres direcciones), pueden ser un nombre, una constante o una variable temporal generada por el compilador. Al final, se está pasando a líneas el **árbol de sintaxis abstracta.**


© Universidad Europea de Madrid. Todos los derechos reservados.

**Código de tres direcciones**

En este ejemplo de evaluación de expresiones se puede ver que **x:= y + z - (u * w)** se transforma en:

```
tmp1:= u*w
tmp2:= z - tmp1
x:= y + tmp2
```

Tenemos tres formas de implementar el código de tres direcciones: **cuartetos**, **tercetos** y **tercetos indirectos.** Por otro lado Louden (2004) define otra forma denominada **código P.**


© Universidad Europea de Madrid. Todos los derechos reservados.

## Cuartetos

Un cuarteto es una estructura con cuatro campos: **(op, b, c, a)** para la asignación a:= b op c, es decir: operación, argumento 1, argumento 2 y resultado.

Cuando hay proposiciones con un solo operador (por ejemplo la asignación de una variable, con signo o no), el argumento 2 no se utiliza.

| Ejemplo con cuartetos: **x:= y + z – (u * w)**, que se transforma en: | Y los cuartetos correspondientes serán: |
|---|---|
| tmp1:= u*w | (*, u, w, tmp1) |
| tmp1:= z – tmp1 | (-, z, tmp1, tmp2) |
| tmp3:= y + tmp2 | (+, y, tmp2, tmp3) |
| x:= tmp3 | (:=, tmp3, , x) |

Los contenidos de los campos argumento1, argumento2 y resultado, dependiendo de la implementación, son punteros a las entradas de la tabla de símbolos.

En el caso de un salto condicional: if x > y then a:= 3;

| | |
|---|---|
| 1 | ( <, x, y, E1) |
| 2 | (goto, E1, , E2) |
| 3 | (:=, 3, , a) |
| 4 | (Etiqueta, , , E2) |

Se evalúa la condición de la expresión booleana (x > y), tal y como vemos en el esquema, y si es cierta se salta a E1 (etiqueta 1) y ejecuta el código correspondiente a la sentencia a: =3, y si es falsa saltamos a E2, que en este caso equivale a continuar con el resto del código del programa.

**Cuarteto**

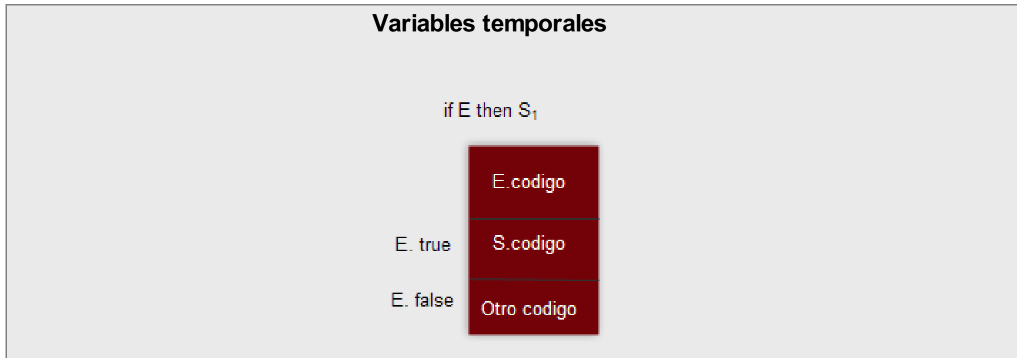
Otros autores lo denominan cuádruplas.

**Contenidos punteros**

Esto incluye también a las **variables temporales,** que deberán insertarse y eliminarse de la tabla de símbolos según se van necesitando.



© Universidad Europea de Madrid. Todos los derechos reservados.






© Universidad Europea de Madrid. Todos los derechos reservados.

## Tercetos

> Los cuartetos, aunque son la herramienta más general, presentan como inconveniente que **ocupan demasiado espacio** y es necesario utilizar muchas variables temporales para los resultados intermedios.

Con el objeto de evitar introducir (y eliminar, si así lo considera el diseñador) las variables temporales en la tabla de símbolos se hace referencia a un valor temporal según la posición de la proposición que lo calcula (Aho et al, 1986).

La estructura de un terceto es (operador, operando 1, operando 2), y el resultado está implícito y asociado a dicho terceto. A su vez los tercetos referencia a otros tercetos que es donde está el resultado que necesitan para operar. Esta estructura es **equivalente a un ASA.**

| Ejemplo con tercetos: **x:= y + z – (u * w),** que se transforma en: | Y los tercetos correspondientes serán: |
|---|---|
| tmp1:= u*w | (*, u, w) |
| tmp2:= z – tmp1 | (-, z, (1)) |
| tmp3:= y + tmp2 | (+, y, (2)) |
| x:= tmp3 | (:=, (3), x) |

En el caso de un salto condicional: if x > y then a:= 3:

| | |
|---|---|
| **1** | (< x, y) |
| **2** | (goto, (1), (4)) |
| **3** | (:=, 3, a) |
| **4** | ( , , ) |

© Universidad Europea de Madrid. Todos los derechos reservados.

## Tercetos indirectos

Son similares a los tercetos, pero incorporan un vector, denominado vector secuencia (VS) que tiene la lista de apuntadores a los tercetos.

| Ejemplo con tercetos indirectos: x:= y + z - (u * w), que se tranforma en: |
| --- |
| tmp1: = u * w |
| tmp2:= z - tmp1 |
| tmp3:= y + tmp2 |
| x: = tmp3 |

| Los tercetos correspondientes serán: |
| --- |
| (*, u, w) |
| (-, z, (1)) |
| (+, y, (2)) |
| (:=, (3), x) |
| VS= (1, 2, 3, 4) |

En este ejemplo no se ve la utilidad porque no hay instrucciones repetidas, pero si las tuviéramos simplemente sería una entrada más en el vector secuencia.

En el caso de un salto condicional: **if x > y then a:= 3**

| | |
| --- | --- |
| 1 | ( <, x, y) |
| 2 | (goto, (1), (4)) |
| 3 | (:=, 3, a) |
| 4 | ( , , ) |
| 5 | VS=(1,2,3) |

Si comparamos entre las tres implementaciones, teniendo en cuenta el nivel de indirección, la facilidad de optimización y el espacio de memoria necesario, vemos que los tercetos indirectos son los que mejor resultado alcanzan.

**Tercetos indirectos**

Presentan como ventaja que son más fáciles de optimizar y que ocupan menos memoria, puesto que si el mismo terceto se repite, se introduce un puntero más al mismo que ya existe.



© Universidad Europea de Madrid. Todos los derechos reservados.

## DDS para expresiones

Una vez que ya sabemos cómo funcionan las distintas implementaciones del código de tres direcciones, necesitamos saber cómo **generar las reglas semánticas correspondientes a las expresiones aritméticas** mediante una **definición dirigida por la sintaxis (DDS),** y que este código intermedio se vaya escribiendo a la vez que realizamos el análisis sintáctico.

Vamos a utilizar un atributo sintetizado, **E.lugar**, que es el nombre que tendrá el valor de E y una función *tempnuevo()*, que devolverá el nombre del siguiente temporal. También vamos a usar la función *escribe( )*, que va a escribir en un fichero el código intermedio que se va generando y utilizaremos la función *busca( )* para buscar si existe un identificador en la tabla de símbolos, si no existe devolverá null.

Reglas semánticas

© Universidad Europea de Madrid. Todos los derechos reservados.

## Evaluación de *arrays* I

En Aho et al (1986), se indica cómo se pueden hacer las operaciones de cálculo de los elementos de un *array*.

### Para un *array* de una dimensión (vector)

Si el **ancho de cada elemento es *a*,** entonces el i-ésimo elemento de la matriz A comienza en la posición: ***base + (i - inf) * a.***

Donde: **inf** es igual al límite inferior de los subíndices (puede empezar en 0 o en 1) y **base** es igual a la dirección relativa de la posición de memoria asignada a la matriz. Es decir, **base es la dirección relativa de A[inf].** Esta expresión se puede evaluar parcialmente durante el proceso de compilación (de forma estática) si se reescribe, a partir de operar (i -inf) * a = i*a - inf*a, quedando: ***i x a + (base - inf * a).***

Denominamos c a la subexpresión *base - inf x a*, quedando *c = base - inf x a* que se puede evaluar cuando aparece la declaración de la matriz.

### Para un *array* de dos dimensiones (matriz)

Hay que tener en cuenta que una matriz bidimensional se puede almacenar de dos formas: fila y columna. Supongamos que lo implementamos utilizando la forma por filas:

- ***base + ((i$_1$ - inf$_1$) * n$_2$ + i$_2$ - inf$_2$) * a,*** donde: ***inf$_1$*** e ***inf$_2$*** son los límites inferiores de los valores de $i_1$ e $i_2$ y ***n$_2$*** es el número de valores que puede tomar $i_2$. Es decir, si $sup_2$ es el límite superior para el valor de $i_2$, entonces $n_2 = sup_2 - inf_2 + 1$.

Suponiendo que $i_1$ e$i_2$ son los únicos valores que no se conocen durante la compilación, la expresión anterior se puede reescribir como:

$$((i_1 * n_2) + i_2) * a + \underbrace{(base - ((inf_1 * n_2) + inf_2) * a}$$

Donde, el término que agrupa el paréntesis se puede determinar durante la compilación. Este cálculo que se ha hecho para dos dimensiones puede ampliarse a n dimensiones (véase Aho et al, 1986).

> **Matriz**
>
> Almacenamos *c* en la entrada de la tabla de símbolos de A, así que la dirección relativa de **A[i]** se obtiene añadiendo simplemente *i* a ac,* cálculo que se tiene que hacer en proceso de ejecución, puesto que a priori no sabemos lo que vale i.



© Universidad Europea de Madrid. Todos los derechos reservados.

**Fila**

En primer lugar todas las columnas de una fila y así sucesivamente cada fila. Esta forma es la que utiliza Pascal.

**Columna**

En primer lugar todas las filas de la primera columna y después el resto de columnas. Esta forma es la que utiliza FORTRAN.



© Universidad Europea de Madrid. Todos los derechos reservados.

## Evaluación de *arrays* II

Ahora necesitamos saber qué código hay que generar para, por ejemplo, una matriz de dos dimensiones, de 5x10 y cuyos límites inferiores de ambas dimensiones es uno, suponiendo que utilizamos dos bytes por cada entero. ¿Cómo será la asignación a:= M [x, y]? Partimos de lo siguiente:

$$((i_1 * n_2) + i_2) * a + (base - ((inf_1 * n_2) + inf_2) * a,$$

1er término

2º término → c:= base - ((1 * 10) + 1) * 2 = base - 22

Puesto que $inf_1$ e $inf_2$ son 1, $n_1$ = 5 y $n_2$ = 10, a = 2:

- $t_1$ := x* 10 /* $i_1 * n_2$ */
- $t_1$ := $t_1$ + y
- $t_2$ := c /* base - 22, es un valor constante que está en la tabla de símbolos */.
- $t_3$ := 2* $t_1$
- $t_4$ := $t_2$[$t_3$]
- x:= $t_4$

Ahora que ya sabemos que hay que obtener, podemos generar el árbol y seguidamente las reglas semánticas, pero primero hay que definir una gramática adecuada para los arrays (Aho et al, 1986).

Para tener disponibles los límites de las distintas dimensiones, es útil reescribir la gramática de la siguiente forma:

- L → lista ] | id.
- lista → lista, E | id [ E.

Para implementar este proceso necesitamos utilizar varios atributos sintetizados.

**Gramática adecuada para los *arrays***

Lo más lógico es la siguiente estructura:

- L → id [ lista ] | id.
- lista → lista, E | E.



© Universidad Europea de Madrid. Todos los derechos reservados.

**Atributos sintetizados**

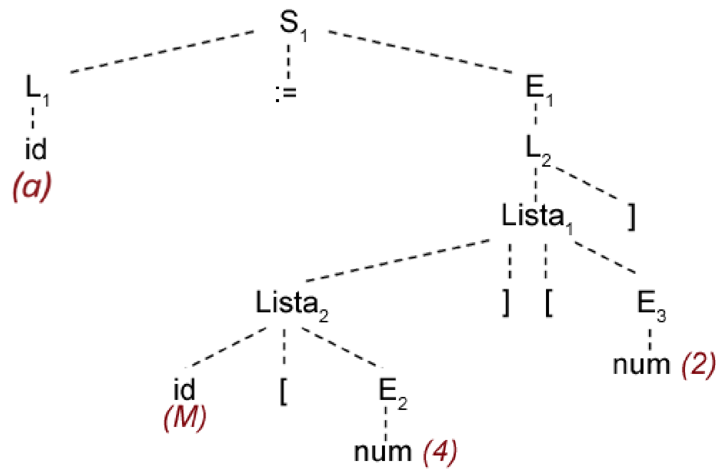
- **L.ndim:** indica la dimensión de la lista o *array*, es en este ejemplo puesto que tiene dos podrá valer 1 o 2.
- **L. lugar:** almacena el valor base (el 2º término) de la lista.
- **L. desplazamiento:** será null cuando sea un identificador simple y distinto de null cuando estemos hablando del *array*. En el ejemplo a.desplazamiento será nulo, pero M.desplazamiento no. Lo simplificaremos por L.desp.



© Universidad Europea de Madrid. Todos los derechos reservados.

## Evaluación de *arrays* III

El árbol tendría la siguiente estructura, para el ejemplo concreto a:= M[4,2].



Reglas semánticas II

Obsérvese cómo se averiguan las dimensiones del array a través de ndim en la producción Lista → id[E y posteriormente se suma uno a ndim en la producción Lista1 → Lista2][E. Se deja como ejercicio a los estudiantes el ver cómo se van rellenando los atributos para el caso concreto a:= M[4,2] según se va avanzando de abajo a arriba por el árbol sintáctico.



© Universidad Europea de Madrid. Todos los derechos reservados.

## Resumen

En este tema hemos aprendido cómo se construye el código de tres direcciones, así como tres formas de implementar el código intermedio, utilizando los **cuartetos, los tercetos y los tercetos indirectos.**

Los cuartetos tienen la estructura de un registro con los siguientes campos: **operación, argumento 1, argumento 2 y resultado.** Mientras que los tercetos dejan el resultado en la dirección del terceto utilizando sólo operación, argumento 1 y argumento 2. Finalmente los tercetos indirectos, que tienen la misma estructura que los tercetos pero se apoyan en un vector secuencia que mantiene el orden de ejecución de los tercetos.

También se ha visto cómo se implementan las expresiones utilizando las definiciones dirigidas por la sintaxis (DDS) y atributos sintetizados.

Posteriormente se ha visto como se evalúan los *arrays* de una y dos dimensiones, calculándose en tiempo de compilación una parte fija, denominada base, la cual se almacena en la tabla de símbolos y a partir de ella y dependiendo de la posición que queramos obtener se produce un desplazamiento donde intervienen los bytes que ocupa el tipo de elementos que contiene la tabla.

Finalmente, se ha realizado un ejercicio completo donde se ha visto como se generan las reglas semánticas para generar el código de tres direcciones necesario para evaluar los *arrays*.


© Universidad Europea de Madrid. Todos los derechos reservados.