

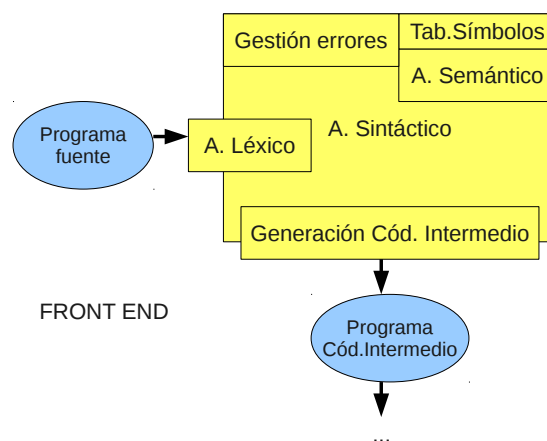
# Tema 6: Generación de código

## Procesamiento de Lenguajes

Dept. de Lenguajes y Sistemas Informáticos  
Universidad de Alicante



## Repaso: estructura estándar de un compilador



Front-end:

- Una pasada: ETDS
- Dos o más pasadas: árbol decorado + recorridos calculando atributos

# Tipos de lenguajes intermedios

$$a = 2.3 + 3 * 4.5 - 7.2 * (3 * 4.5)$$

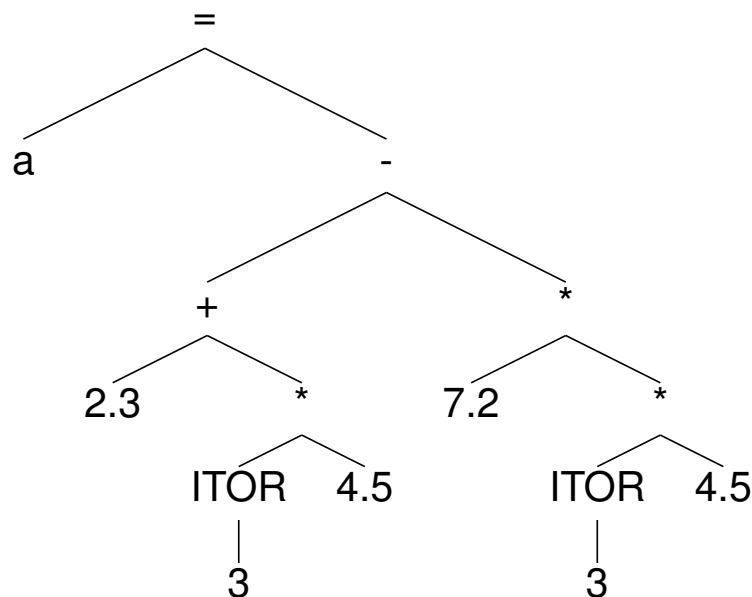
- Código de tres direcciones

```
ITOR 3 t1
MULR t1 4.5 t2
ADDR 2.3 t2 t3
ITOR 3 t4
MULR t4 4.5 t5
MULR 7.2 t5 t6
SUBR t3 t6 t7
STOR t7 a
```

## Tipos de lenguajes intermedios (2)

$$a = 2.3 + 3 * 4.5 - 7.2 * (3 * 4.5)$$

- Árbol sintáctico abstracto (*abstract syntax tree*, AST)



## Tipos de lenguajes intermedios (3)

$$a = 2.3 + 3 * 4.5 - 7.2 * (3 * 4.5)$$

- Máquina virtual pseudo-ensamblador (p.ej. m2r, con acumulador)

```
mov $2.3 100
mov #3 101
mov $4.5 102
mov 101 A      ; convertir '3' a real
itor
mov A 103
mov 103 A
mulr 102      ; multiplicar '3.0' por '4.5'
mov A 104
mov 100 A
addr 104      ; sumar '2.3' con '3.0*4.5'
mov A 105
...
```

## Tipos de lenguajes intermedios (4)

$$a = 2.3 + 3 * 4.5 - 7.2 * (3 * 4.5)$$

- Máquina virtual de pila, como P-code (usado en los primeros compiladores de Pascal)

```
LOADI dir(a)
LOADR 2.3
LOADI 3
ITOR
LOADR 4.5
MULR
ADDR
LOADR 7.2
LOADI 3
ITOR
LOADR 4.5
MULR
MULR
SUBR
STOR
```

4.5
4.533.013.5
33.013.57.297.2
2.315.8-81.4
dir(a)

## Tipos de lenguajes intermedios (5)

$$a = 2.3 + 3 * 4.5 - 7.2 * (3 * 4.5)$$

- Máquina virtual de pila, como CIL (lenguaje intermedio de la plataforma .Net)

```
ldc.r8 2.3
ldc.i4 3
conv.r8
ldc.r8 4.5
mul
add
ldc.r8 7.2
ldc.i4 3
conv.r8
ldc.r8 4.5
mul
mul
sub
stloc 'a'
```

## Tipos de lenguajes intermedios (6)

```
float a, b=4.5, c=7.2; a = 2.3 + 3*b - c*(3*b);
```

- Máquina virtual de LLVM

```
%1 = alloca float, align 4
%2 = alloca float, align 4
%3 = alloca float, align 4
store float 4.500000e+00, float* %2, align 4
store float 0x401CCCCC00000000, float* %3, align 4
%4 = load float, float* %2, align 4
%5 = fmul float 3.000000e+00, %4
%6 = fpext float %5 to double
%7 = fadd double 2.300000e+00, %6
%8 = load float, float* %3, align 4
%9 = load float, float* %2, align 4
%10 = fmul float 3.000000e+00, %9
%11 = fmul float %8, %10
%12 = fpext float %11 to double
%13 = fsub double %7, %12
%14 = fptrunc double %13 to float
store float %14, float* %1, align 4
```

# Sistema de tipos

- Cada lenguaje fuente tiene un sistema de tipos, que establece qué mezclas de tipos están permitidas y qué conversiones es necesario realizar
  - ▶ En Pascal solamente se pueden mezclar enteros y reales en las expresiones, pero no booleanos ni caracteres. **No se permite asignar un valor real a una variable entera**
  - ▶ En C se permiten todas las combinaciones, pero algunas generan *warning* (que no deben ignorarse). Por ejemplo:

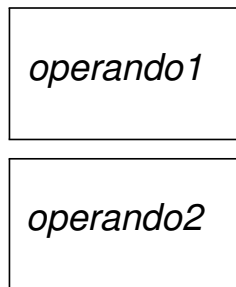
```
int a = '0' * 2 + 3.9;
// >> 48 * 2 + 3.9 >> 96+3.9 >> 96.0+3.9 >> 99.9 >> 99
```
- El compilador debe calcular el tipo de cada subexpresión, generar las conversiones necesarias, y producir errores si el sistema de tipos no permite alguna mezcla (p.ej. `true + 2` en Pascal)

## Generación de código para expresiones en `m2r`

- En `m2r` se utiliza el acumulador y variables temporales para realizar las operaciones
- Operaciones aritméticas: `addi`, `addr`, `subi`, `subr`, `muli`, `mulr`, `divi`, `divr`, ...
- Todos los operadores binarios en `m2r` toman el acumulador como primer operando, y lo operan con el argumento del operador
- El tipo de los dos operandos debe ser el mismo (entero o real).
- Operaciones de conversión (sobre el acumulador): `itor`, `rtoi`
- **IMPORTANTE:** el resultado de cualquier operación debe almacenarse en una variable temporal (casi siempre)

## Generación de código para expresiones en $m2r$ (2)

- Los pasos que hay que realizar para hacer una operación son:
  - 1 Almacenar el primer operando (que está en una variable temporal) en el acumulador
  - 2 Operar con el segundo operando (que también está en una temporal)
  - 3 Almacenar el resultado, que está en el acumulador, en una variable temporal



```
mov  operando1.dirtmp  A
op    operando2.dirtmp
mov   A    nuevatmp
```

## Generación de código para expresiones en $m2r$ (3)

### Ejemplo: $2 * 3 + 4$

```
mov #2 16100 ; el 2 a una temporal
mov #3 16101 ; el 3 a una temporal
mov 16100 A ; el 2 al acumulador
mul 16101 ; multiplicar por 3
mov A 16102 ; guardar el resultado en temporal
mov #4 16103 ; el 4 a una temporal
mov 16102 A ; el 6 al acumulador
add 16103 ; sumar con 4
mov A 16104 ; guardar el resultado en temporal
```

## ETDS para generar código m2r para expresiones (1)

```
T  →  numentero { T.tipo := ENTERO;
                  tmp := NuevaTemporal();
                  T.cod := mov #||numentero.lexema||tmp;
                  T.dir := tmp }
T  →  numreal { T.tipo := REAL;
                tmp := NuevaTemporal();
                T.cod := mov $||numreal.lexema||tmp;
                T.dir := tmp }
T  →  id { if((simbolo = tsActual->buscar(id.lexema)) == null)
          errorSemantico(...)
        else
          tmp := NuevaTemporal();
          T.cod = mov ||simbolo.posicion||tmp;
          T.tipo := simbolo.tipo;
          T.dir := tmp
        endif }
E  →  T { E.tipo := T.tipo; E.cod := T.cod; E.dir := T.dir }
```

## ETDS para generar código m2r para expresiones (2)

```
E  →  E opas T {
      tmp := NuevaTemporal(); E.dir := tmp;
      if(E1.tipo == ENTERO && T.tipo == ENTERO)
        E.cod := E1.cod||T.cod||mov E1.dir A||
               opas.trad||i ||T.dir||mov A ||tmp;
        E.tipo := ENTERO
      elsif(E1.tipo == REAL && T.tipo == ENTERO)
        tmpcnv := NuevaTemporal();
        E.cod := E1.cod||T.cod||mov T.dir A||
               itor ||mov A tmpcnv||mov E1.dir A||
               opas.trad||r ||tmpcnv||mov A ||tmp;
        E.tipo := REAL
      elsif(E1.tipo == ENTERO && T.tipo == REAL)
        E.cod := E1.cod||T.cod||mov E1.dir A||
               itor ||opas.trad||r ||T.dir||mov A ||tmp;
        E.tipo := REAL
      else // REAL && REAL
        E.cod := E1.cod||T.cod||mov E1.dir A||
               opas.trad||r ||T.dir||mov A ||tmp;
        E.tipo := REAL
      endif
    }
```

(el atributo **opas.trad** será *add* o *sub*, según el lexema de **opas**)

# Operadores relacionales en `m2r`

Las instrucciones en `m2r` para los operadores relacionales son:

OPERADOR	INSTRUCCIÓN
<code>==</code>	<code>eqli/eqlr</code>
<code>!=</code>	<code>neqi/neqr</code>
<code>&gt;</code>	<code>gtri/gtrr</code>
<code>&gt;=</code>	<code>geqi/geqr</code>
<code>&lt;</code>	<code>lssi/lssr</code>
<code>&lt;=</code>	<code>leqi/leqr</code>

Como en los operadores aritméticos, los dos operandos deben ser del mismo tipo, entero o real. El resultado es siempre un valor entero, un 0 o un 1.

## Operadores booleanos

- Los operadores booleanos trabajan con los dos valores booleanos, *cierto* y *falso*.
- En algunos lenguajes como C y C++, se asume que un 0 es *falso*, y cualquier valor distinto de 0 es *cierto*, mientras que en lenguajes como Pascal solamente se puede usar `true` y `false` (los operadores relacionales generan un valor booleano). Al generar código intermedio debe tenerse en cuenta esta cuestión ya que el código que se debe generar en ambos casos puede ser diferente, dependiendo de las instrucciones del lenguaje intermedio.
- **IMPORTANTE:** en general, es recomendable que los valores booleanos se representen internamente en el código intermedio con los valores 0 y 1.



## Operadores booleanos (2)

Dada una expresión  $A \text{ op } B$ , hay dos formas de evaluar los operadores AND y OR:

- 1 Evaluación similar a la de otros operadores binarios (como p.ej. en Pascal): se evalúa  $A$ , se evalúa  $B$ , y se evalúa la operación AND u OR. Las instrucciones en `m2r` para los operadores booleanos son:

OPERADOR	INSTRUCCIÓN
AND	andi/andr
OR	ori/orr
NOT	noti/notr

- 2 Evaluación en cortocircuito:

**AND** : se evalúa  $A$ , y solamente si el resultado es *cierto* se evalúa  $B$  (si  $A$  es *falso* no vale la pena evaluar  $B$ , el resultado va a ser *falso*)

**OR** : se evalúa  $A$ , y si es *falso* se evalúa  $B$  (si  $A$  es *cierto* el resultado va a ser *cierto*)

La implementación de la evaluación en cortocircuito se realiza con saltos condicionales, casi como una instrucción condicional:

$A \ \&\& \ B \Rightarrow \text{if } A \text{ then } B$   
 $A \ || \ B \Rightarrow \text{if } A \text{ then cierto else } B$

## Generación de código `m2r` para instrucciones (1)

- Asignación:

$\text{Instr} \longrightarrow \text{id asig Expr}$

*Expr.cod*

conversiones? (itor/rtoi)

`mov Expr.dir id.posicion`

Ejemplo:  $a = 2 * 3$

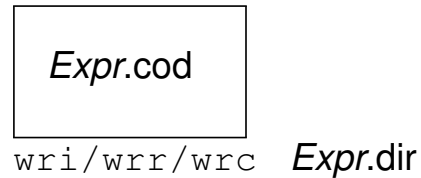
```
mov #2 16100
mov #3 16101
mov 16100 A
mul 16101
mov A 16102 ; Expr.cod, Expr.dir=16102
mov 16102 100 ; a.posicion = 100
```

- ▶ **IMPORTANTE:** el código generado para las expresiones deja el valor de la expresión en la temporal *Expr.dir*
- ▶ Dependiendo del lenguaje, puede ser necesario hacer conversiones entre tipos o bien producir errores semánticos
- ▶ Si hay que hacer conversiones, es posible que se tenga que utilizar una nueva variable temporal

## Generación de código $m2r$ para instrucciones (2)

- Salida

$Instr \longrightarrow \mathbf{write} \ Expr$



- ▶ Dependiendo del tipo de la expresión, es necesario utilizar la instrucción de escritura correspondiente
- ▶ Según la semántica del lenguaje fuente, es posible que después de escribir la expresión se tenga que escribir un “\n”, para lo que se debe usar la instrucción *wrl*

## Generación de código $m2r$ para instrucciones (3)

- Entrada

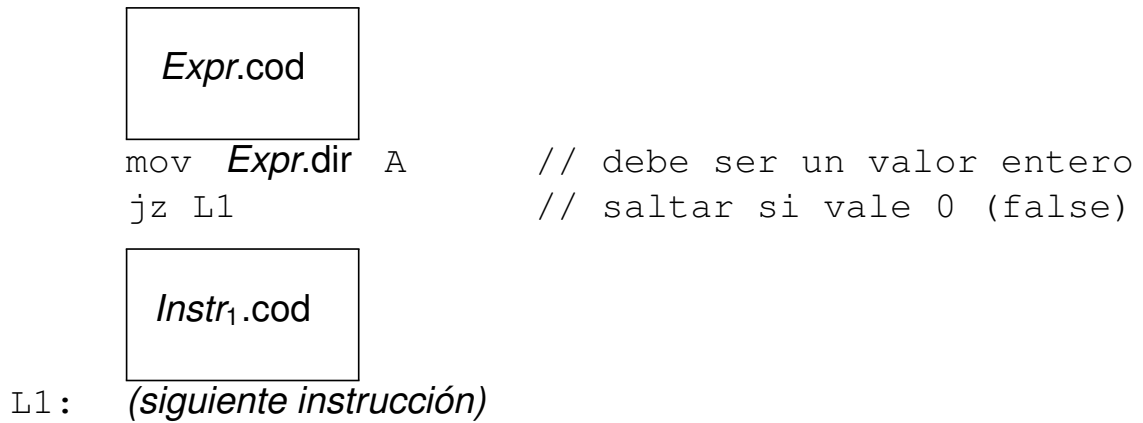
$Instr \longrightarrow \mathbf{read} \ id$

- ▶ Si la variable es de tipo entero:  
*rdi id.posicion*
- ▶ Si la variable es de tipo real:  
*rdr id.posicion*
- ▶ Si la variable es de tipo carácter (o booleano?):  
*rdc id.posicion*

## Generación de código $m2r$ para instrucciones (4)

- Condicional

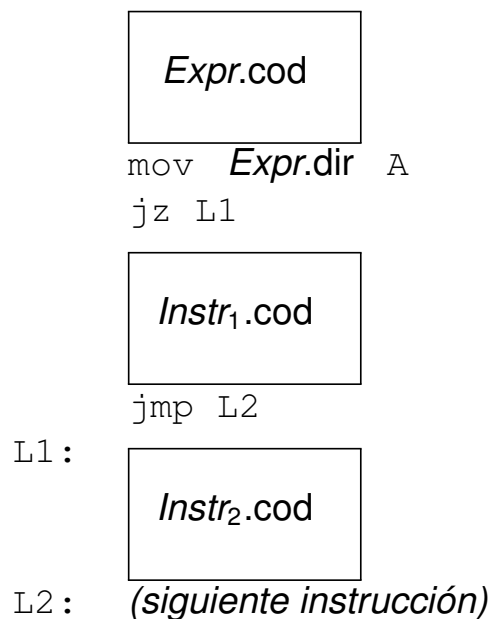
$Instr \longrightarrow \text{if } ( Expr ) Instr_1$



## Generación de código $m2r$ para instrucciones (5)

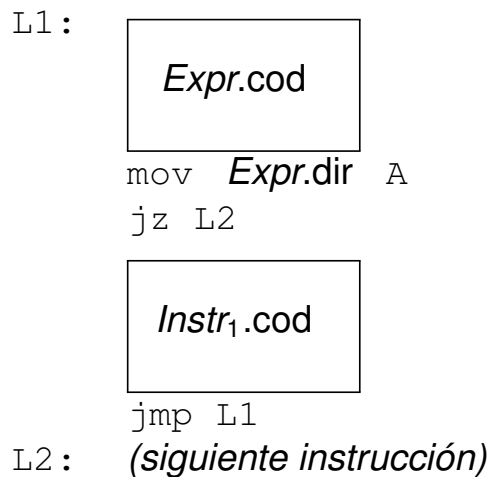
- Condicional (2)

$Instr \longrightarrow \text{if } ( Expr ) Instr_1 \text{ else } Instr_2$



- Iteración

$Instr \longrightarrow \text{while } ( Expr ) Instr_1$



## Ejercicio 1

Indica qué código se generaría en  $m2r$  para las instrucciones de iteración siguientes:

- 1 for de C, C++, Java, ...

$Instr \longrightarrow \text{for } ( Expr_1 ; Expr_2 ; Expr_3 ) Instr_1$

La expresión  $Expr_1$  se ejecuta una vez al principio del bucle, la expresión  $Expr_2$  se ejecuta en cada paso del bucle, y si el resultado es cierto se ejecuta el código de la instrucción, y la expresión  $Expr_3$  se ejecuta después del código de la instrucción en cada paso del bucle.

- 2 do-while de C, C++, Java, ...

$Instr \longrightarrow \text{do } Instr_1 \text{ while } ( Expr )$

La instrucción se ejecuta al menos una vez, y se repite mientras la expresión sea cierta.

- 3 repeat-until de Pascal

$Instr \longrightarrow \text{repeat } Instr_1 \text{ until } ( Expr )$

La instrucción se ejecuta al menos una vez, y se repite hasta que la expresión sea cierta.

# Tipos compuestos: arrays

- Los vectores son *arrays* unidimensionales, las matrices son *arrays* multidimensionales, pero se tratan de la misma manera
- Existen básicamente dos formas de declarar *arrays*:
  - 1 Al estilo de C: `int a[10]` (los índices del *array* van de 0 a 9)
  - 2 Al estilo de Pascal: `a:array [1..10] of integer` (los índices del *array* van de 1 a 10, obviamente)
- Los *arrays* multidimensionales se pueden ver como *arrays* de *arrays*:

`int a[10][15]   ≡   array [0..9] of array [0..14] of integer`

- Para almacenar la información de los *arrays* (y otros tipos) se utiliza una **tabla de tipos**

## Tipos compuestos: arrays (2)

Tabla de tipos (estilo C):

```
int main() {  
    int a[10],b[7][5];  
    double c[15][25][35];  
  
    ...  
}
```

	TIPO	TAMAÑO	TIPO BASE
1	ENTERO		
2	REAL		
3	ARRAY	10	1
4	ARRAY	5	1
5	ARRAY	7	4
6	ARRAY	35	2
7	ARRAY	25	6
8	ARRAY	15	7

NOMBRE	TIPO	DIRECCIÓN
a	3	0
b	5	10
c	8	45

## Tipos compuestos: arrays (3)

Tabla de tipos (estilo Pascal):

```
program p;  
var  
  a:array [1..10] of integer;  
  b:array [10..16,21..25]  
    of integer;  
  c:array [1..15,1..25,1..35]  
    of real;  
  
  ...  
end.
```

	TIPO	LIM. INF.	LIM. SUP.	TIPO BASE
1	ENTERO			
2	REAL			
3	ARRAY	1	10	1
4	ARRAY	21	25	1
5	ARRAY	10	16	4
6	ARRAY	1	35	2
7	ARRAY	1	25	6
8	ARRAY	1	15	7

Es necesario almacenar el límite inferior y superior del rango, aunque también se puede almacenar el límite inferior y el tamaño (o incluso los tres valores)

## Tipos compuestos: arrays (4)

ETDS para guardar *arrays* en la tabla de tipos

---

$D \rightarrow T \text{ id } \{tsActual \rightarrow \text{nuevoSimb}(\text{id.lexema}, T.tipo, T.tam);$   
                   $L.th := T.tipo; L.tah := T.tam\} \quad L$

$T \rightarrow \text{real } \{T.tipo := \text{REAL}; T.tam := 1\}$

$T \rightarrow \text{entero } \{T.tipo := \text{ENTERO}; T.tam := 1\}$

$T \rightarrow \text{tabla num de } T \{T.tipo := \text{ttipos} \rightarrow \text{nuevoTipo}(\text{num.lexema}, T_1.tipo);$   
                           $T.tam := \text{num.valor} * T_1.tam\}$

$L \rightarrow , \text{ id } \{tsActual \rightarrow \text{nuevoSimb}(\text{id.lexema}, L.th, L.tah);$   
                   $L_1.th := L.th; L_1.tah := L.tah\} \quad L$

$L \rightarrow \epsilon$

## Tipos compuestos: arrays (5)

Los *arrays* se almacenan de forma lineal, como un vector

```
int a[4][3];
```

```
a[2][1] = 7;
```

```
a[0][2] = 5;
```

```
a[3][0] = 8;
```

a[0]			a[1]			a[2]			a[3]		
		5					7		8		

## Tipos compuestos: arrays (6)

Generación de código para acceder a posiciones de *arrays*:

- Cálculo de la dirección de memoria:

```
int a[10][20][30];
```

```
...
```

```
... a[i][j][k] ...
```

$$\begin{aligned} \text{dir}(a[i][j][k]) = & \text{dir}(a) + i \times (20 \times 30 \times \text{sizeof}(\text{int})) \\ & + j \times (30 \times \text{sizeof}(\text{int})) \\ & + k \times \text{sizeof}(\text{int}) \end{aligned}$$

- Se suele utilizar una formulación recursiva:

```
t1 := 0 // base de la recursión
t2 := t1 × 10 + i
t3 := t2 × 20 + j
t4 := t3 × 30 + k
t5 := dir(a) + t4 × sizeof(int) // paso final
```

## Tipos compuestos: arrays (7)

Generación de código para acceder a posiciones de *arrays* (estilo Pascal):

- Cálculo de la dirección de memoria:

```
var a:array [1..10,15..20,1234..1244] of integer;
...
... a[i,j,k] ...
dir(a[i,j,k]) = dir(a) + (i-1) × (6 × 11 × sizeof(integer))
                + (j-15) × (11 × sizeof(integer))
                + (k-1234) × sizeof(integer)
```

- Con la formulación recursiva:

```
t1 := 0 // base de la recursión
t2 := t1 × 10 + i - 1
t3 := t2 × 6 + j - 15
t4 := t3 × 11 + k - 1234
t5 := dir(a) + t4 × sizeof(integer) // paso final
```

(nota: para restar 1, 15 y 1234 no se necesitan más temporales en m2r)

## Tipos compuestos: arrays (8)

ETDS para acceder a posiciones de *arrays* (1)

```
R → id {if((simbolo = tsActual->buscar(id.lexema)) == null)
        errorSemantico(...)
      else
        tmp := NuevaTemporal(); R.dir := tmp
        R.cod = mov #0 ||tmp;
        R.tipo := simbolo.tipo;
        R.dbase := simbolo.posicion;
        R.dir := tmp}
R → R [ {if(!esArray(R1.tipo))errorSemantico(...)}
      E ] {if(E.tipo != ENTERO)errorSemantico(...)
      else
        R.tipo := ttipos->tipoBase(R1.tipo);
        R.dbase := R1.dbase;
        tmp := NuevaTemporal(); R.dir := tmp;
        R.cod = R1.cod||E.cod||
          mov ||R1.dir|| A||
          muli #||ttipos->tamaño(R1.tipo)||
          addi ||E.dir||
          mov A ||tmp; }
```



# Tipos compuestos: arrays (9)

## ETDS para acceder a posiciones de *arrays* (2)

```
F → R {if(esArray(R.tipo))
      errorSemantico(...)
    else
      tmp := NuevaTemporal(); F.dir := tmp
      F.cod := R.cod||
        mov ||R.dir|| A
        muli #||sizeof(R.tipo)||
        addi #||R.dbase||
        mov @A ||tmp;
      F.tipo := R.tipo; }
I → R asig {if(esArray(R.tipo))errorSemantico(...)}
E {// comprobaciones semánticas Ref := E
  I.cod := R.cod||E.cod||
    mov ||R.dir|| A
    muli #||sizeof(R.tipo)||
    addi #||R.dbase||
    mov ||E.dir|| @A}
```

## Un ejemplo: tablas de símbolos y tipos

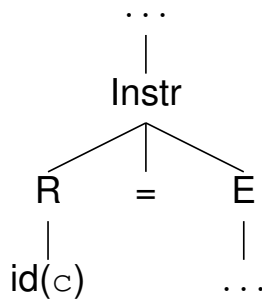
```
int main() {
  int i,j,a[4][3];
  int c;
  float ff;

  ...
}
```

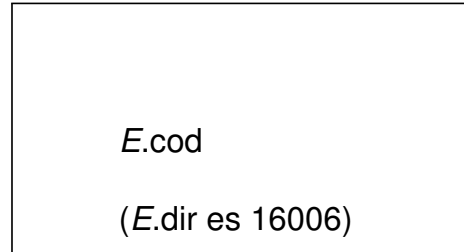
NOMBRE	TIPO	POSICIÓN	...
i	1	0	...
j	1	1	...
a	4	2	...
c	1	14	...
ff	2	15	...

	TIPO	TAMAÑO	TIPO BASE
1	ENTERO		
2	REAL		
3	ARRAY	3	1
4	ARRAY	4	3

## Un ejemplo (2): código para $c = a[2][1];$

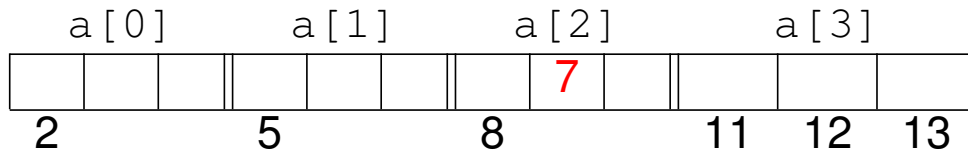


mov #0 16000 ; t1 = 0 (de c =



mov 16000 A  
 addi #14 ; t1(c) + dir(c)  
 mov 16006 @A ; c = a[2][1]

## Un ejemplo (3): código para $c = a[2][1];$



$dir(a) = 2$   
 $dir(c) = 14$

$t_1 := 0$   
 $t_2 := t_1 \times 4 + 2$   
 $t_3 := t_2 \times 3 + 1$   
 $t_4 := t_3 + 2$

mov #0 16000 ; t1 = 0 (de c =  
 mov #0 16001 ; t1 = 0 (de a[  
 mov #2 16002 ; el 2 de [ 2 ]  
 mov 16001 A  
 muli #4  
 addi 16002  
 mov A 16003 ; t2 = t1 \* 4 + 2  
 mov #1 16004 ; el 1 de [ 1 ]  
 mov 16003 A  
 muli #3  
 addi 16004  
 mov A 16005 ; t3 = t2 \* 3 + 1  
 mov 16005 A  
 addi #2 ; t4 = t3 + dir(a)=2  
 mov @A 16006  
 mov 16000 A  
 addi #14 ; t1(c) + dir(c)  
 mov 16006 @A ; c = a[2][1]

# Tipos compuestos: registros/clases

- Los registros (o clases) suelen tener su propia tabla de símbolos para almacenar los campos (o atributos) del registro, y en la tabla de tipos se guarda un enlace a dicha tabla de símbolos:

```
struct {  
    int dni;  
    char letra;  
    double sueldo;  
} empleado;
```

- La generación de código no es aparentemente difícil:

```
empleado.dni        dir(empleado) + 0  
empleado.letra      dir(empleado) + sizeof(int)  
empleado.sueldo     dir(empleado) + sizeof(int) + sizeof(char)
```

En este ejemplo, los campos del registro se pueden tratar como variables (su dirección es fija y conocida en tiempo de compilación).

## Tipos compuestos: registros/clases (2)

### Problema: arrays de registros

```
struct {  
    int dni;  
    char letra;  
    double sueldo;  
} empleado[MAXEMPL];  
  
...  
  
empleado[i].letra
```

La dirección donde comienza el registro  $i$ -ésimo no es conocida en tiempo de compilación, luego se debe generar código para calcular la dirección del registro, y para luego sumarle `sizeof(int)`

**Más difícil todavía:** arrays de registros que tienen campos que son arrays de registros, p. ej. `a[i].b[j].c`

# Tipos compuestos: registros/clases (3)

Código aproximado que se debería generar para:

```
print a[i+1].b[j].c
```

```
mov #dir(a) t1 ; dirección base de "a"
mov dir(i) t2
mov #1 t3
mov t2 A
addi t3
mov A t4 ; código de la expresión "i+1"
mov t4 A ; desplazamiento índice array "a"
addi t1 ; sumar dirección base array "a"
addi #dir(b) ; sumar dirección relativa de b
mov A t5 ; dirección base de a[i+1].b
mov dir(j) t6 ; código de la expresión "j"
mov t6 A ; desplazamiento índice array "b"
addi t5 ; sumar dirección base array "b"
addi #dir(c) ; sumar dirección relativa de c
mov A t8 ; dirección base de a[i+1].b[j].c
mov t8 A
mov @A t9 ; acceso final a a[i+1].b[j].c
wrc t9
```

## Ejercicio 2

Diseña un ETDS que genere código  $m2r$  para el siguiente fragmento de gramática:

$I$	$\longrightarrow$	<b>print</b> $T$
$T$	$\longrightarrow$	$T$ <b>opas</b> $F$
$T$	$\longrightarrow$	$F$
$T$	$\longrightarrow$	<b>opas</b> $F$
$F$	$\longrightarrow$	<b>numeroentero</b>
$F$	$\longrightarrow$	$R$
$F$	$\longrightarrow$	$( T )$
$R$	$\longrightarrow$	<b>id</b> $D$
$D$	$\longrightarrow$	$D [ T ]$
$D$	$\longrightarrow$	$D \{ T \}$
$D$	$\longrightarrow$	$\epsilon$

Ten en cuenta que se permite acceder a posiciones de arrays de dos formas:

- 1 con corchetes, "[ ]", en cuyo caso el acceso será el normal en lenguajes como C/C++, donde la posición 0 es la primera del array.
- 2 con llaves, "{ }", en cuyo caso el acceso será desde el final del array, con números negativos. Por ejemplo, en un vector "v" de tamaño 10, la referencia "v{0}" se refiere a la última la posición del vector (sería equivalente a "v[9]"), y la referencia "v{-2}" sería la antepenúltima (equivalente a "v[7]").

## Ejercicio 3

Diseña un ETDS que genere código  $m2r$  para el siguiente fragmento de gramática:

$I$	$\longrightarrow$	<b>print</b> $T$
$T$	$\longrightarrow$	$T$ <b>opas</b> $F$
$T$	$\longrightarrow$	$F$
$T$	$\longrightarrow$	<b>opas</b> $F$
$F$	$\longrightarrow$	<b>numeroentero</b>
$F$	$\longrightarrow$	$R$
$F$	$\longrightarrow$	<b>(</b> $T$ <b>)</b>
$R$	$\longrightarrow$	<b>id</b> $A$
$A$	$\longrightarrow$	$\epsilon$
$A$	$\longrightarrow$	<b>[</b> $D$ <b>]</b>
$D$	$\longrightarrow$	$D$ <b>coma</b> $T$
$D$	$\longrightarrow$	$T$

En este lenguaje, los arrays se han declarado al estilo de Pascal, con límite inferior y superior (p.ej.  $a[7..15, 3..9]$ ).

Puedes utilizar las funciones/métodos que necesites para acceder a dichos límites en la tabla de tipos, pero no debes generar código para comprobar que el índice está dentro de los límites.