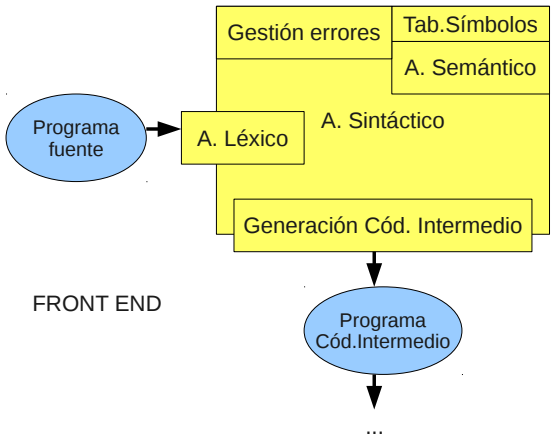
Tema 6: Generación de código

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)
```

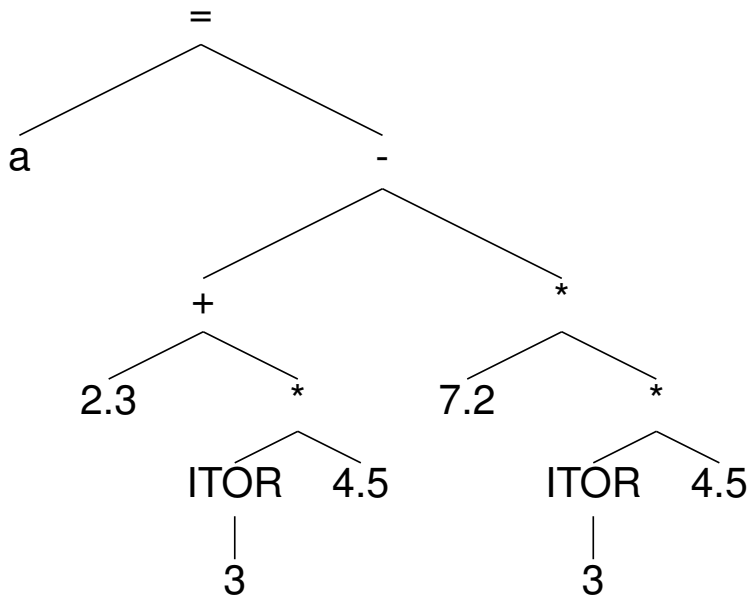
- Código de tres direcciones

```
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)
```

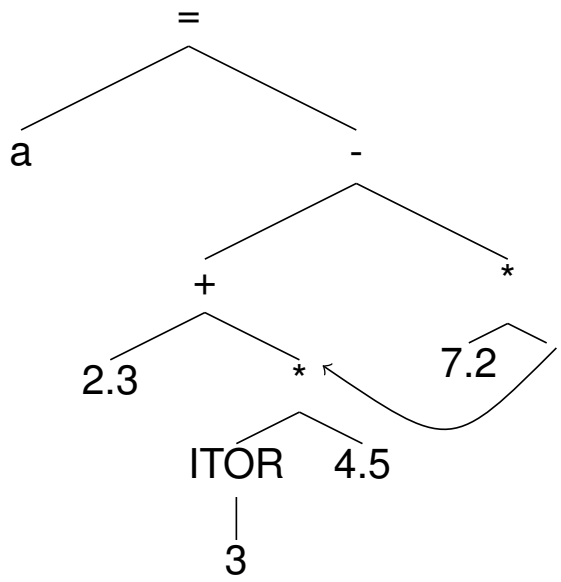
- Árbol sintático abstracto (*abstract syntax tree*, AST)



Tipos de lenguajes intermedios (3)

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

- Grafo dirigido acíclico (*directed acyclic graph*, DAG)



Tipos de lenguajes intermedios (4)

```
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 (5)

```
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
```


Tipos de lenguajes intermedios (6)

```
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'
```

Tabla de símbolos

- En la tabla de símbolos se almacena información de cada símbolo que aparece en el programa fuente:
  - variable: nombre, tipo, dirección/posición, tamaño, ...
  - método/función: nombre, tipo devuelto, tipo y nº de los parámetros, etiqueta de comienzo del código, ...
  - tipos definidos por el usuario: nombre, tipo, tamaño, ...
  - otros ...
- Ejemplo:

```
int main() {
 int a,b;
 double c,d;

 ...
}
```

NOMBRE	TIPO	POSICIÓN	...
a	ENTERO	0	...
b	ENTERO	1	...
c	REAL	2	...
d	REAL	3	...

Tabla de símbolos (2)

Las operaciones que se suelen hacer con la tabla de símbolos son:

- `nuevoSimbolo` : añadir un nuevo símbolo al final de la tabla, comprobando previamente que no se ha declarado antes
- `buscarSimbolo` : buscar un símbolo en la tabla para ver si se ha declarado o no, y obtener su información

Implementación:

- Se suele utilizar una tabla hash, es muy eficiente para el almacenamiento de identificadores
- La función de búsqueda, `buscarSimbolo`, debería devolver toda la información del símbolo (un registro de tipo Simbolo), de manera que el acceso a cada dato del símbolo no implicara una nueva búsqueda en la tabla

Tabla de símbolos (3)

La gestión de la tabla se complica con ámbitos anidados:

```
int f()
{
  int a,c=7;

  {
    double a,b;

    a = 7.3+c;  // 'a' es real , 'c' es del ámbito anterior
  }
  a = 5;       // 'a' es entera
  b = 3.5;     // error, 'b' ya no existe
}
```

- Al principio de cada bloque se abre un nuevo ámbito, en el que se pueden declarar símbolos con el mismo nombre que en los ámbitos anteriores abiertos (pero no con el mismo nombre que otros símbolos del ámbito)
- Cuando se acaba el bloque, se deben *olvidar* las variables declaradas en ese ámbito.

Tabla de símbolos (4)

Implementación de la tabla de símbolos con ámbitos anidados:

1. Usar un vector de símbolos, marcando y guardando el comienzo de cada ámbito, de forma que las operaciones `nuevoSimbolo` y `buscarSimbolo` empiecen la búsqueda por el final, y paren al principio del ámbito (`nuevoSimbolo`) o sigan hacia el principio del vector (`buscarSimbolo`)
2. Usar una especie de pila de tablas de símbolos: cada tabla de símbolos almacena en sus datos internos una referencia a la tabla de símbolos del ámbito *padre*. En `buscarSimbolo`, si no se encuentra un símbolo en la tabla actual, se busca recursivamente en las tablas de los ámbitos abiertos anteriores.

Tabla de símbolos (5)

ETDS para gestionar la tabla de símbolos

$D \rightarrow T$ **id** {`tsActual->nuevoSimbolo`(**id**.*lexema*, *T.tipo*); *L.th* := *T.tipo*} *L*

$T \rightarrow$ **float** {*T.tipo* := *REAL*}

$T \rightarrow$ **int** {*T.tipo* := *ENTERO*}

$L \rightarrow$ **,** **id** {`tsActual->nuevoSimbolo`(**id**.*lexema*, *L.th*); L_1.*th* := *L.th*} *L*

$L \rightarrow \epsilon$

...

Instr → **id** {*if*((*simbolo* = `tsActual->buscarSimbolo`(**id**.*lexema*)) == *null*)
`errorSemantico(...)`}
asig *Expr* { ...
Instr.trad := *Expr.trad*|| ... ||`mov` *Expr.dir* *simbolo*.`posicion`
}

...

Factor → **id** {*if*((*simbolo* = `tsActual->buscarSimbolo`(**id**.*lexema*)) == *null*)
`errorSemantico(...)`
else
tmp := *NuevaTemporal*(); *Factor.dir* := *tmp*;
Factor.trad := `mov` *simbolo*.`posicion` *tmp*;
Factor.tipo := *simbolo*.`tipo`
endif}

Tabla de símbolos (6)

ETDS para gestionar los ámbitos

S → {`tsActual = nuevaTablaSimbolos(NULL)`} *SecSp*

...

Sp → *TipoFuncion* **id** **(**
{`tsActual->nuevoSimbolo`(**id**.*lexema*, *TipoFuncion.tipo*)
`tsActual = nuevaTablaSimbolos(tsActual)`}
Args **)** *Bloque*
{ ...
`tsActual = tablaPadre(tsActual)`
...
}

...

Instr → { `tsActual = nuevaTablaSimbolos(tsActual)`}
Bloque { ...
`tsActual = tablaPadre(tsActual)`
...
}

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)

ETDS para el cálculo de tipos en expresiones

$E \longrightarrow E$ **opas** T {
  if(E_1.*tipo* == *ENTERO* && *T.tipo* == *ENTERO*)
    E.tipo := *ENTERO*
  elsif(E_1.*tipo* == *REAL* && *T.tipo* == *ENTERO*)
    E.tipo := *REAL*
  elsif(E_1.*tipo* == *ENTERO* && *T.tipo* == *REAL*)
    E.tipo := *REAL*
  else // REAL && REAL
    E.tipo := *REAL*
  endif }

$E \longrightarrow T$ {*E.tipo* := *T.tipo*}

$T \longrightarrow$ **numentero** {*T.tipo* := *ENTERO*}

$T \longrightarrow$ **numreal** {*T.tipo* := *REAL*}

$T \longrightarrow$ **id** {*if*((*simbolo* = `tsActual->buscarSimbolo`(**id**.*lexema*)) == *null*)
  `errorSemantico`(...)
  else
    ...
    T.tipo := *simbolo*.`tipo`
  endif }

ETDS para el cálculo de tipos en expresiones (2)

Otra forma de hacerlo (que no utilizaremos):

$E \longrightarrow E$ **opas** T {
  if(E_1.*tipo* == *ENTERO* && T.*tipo* == *ENTERO*)
    E.*tipo* := *ENTERO*
  else // cualquier otra combinación
    E.*tipo* := *REAL*
  }

$E \longrightarrow T$ {E.*tipo* := T.*tipo*}

$T \longrightarrow$ **numentero** {T.*tipo* := *ENTERO*}

$T \longrightarrow$ **numreal** {T.*tipo* := *REAL*}

$T \longrightarrow$ **id** {*if*((*simbolo* = tsActual->buscarSimbolo(**id**.*lexema*)) == *null*)
    errorSemantico(...)
  else
    ...
    T.*tipo* := *simbolo*.tipo
  endif}

Generación de código para expresiones en m2r

- En m2r se utiliza el acumulador y variables temporales para realizar las operaciones
- Los pasos que hay que realizar para hacer una operación son:
  1. Almacenar el primer operando en el acumulador: mov *operando1* A
  2. Operar con el segundo operando: op *operando2*
  3. Almacenar el resultado, que está en el acumulador, en una variable temporal: mov A *temporal*
- Operaciones aritméticas: addi, addr, subi, subr, muli, mulr, divi, divr, ...
- Operaciones de conversión (sobre el acumulador): itor, rtoi

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

$T \longrightarrow$ **numentero** { $T.tipo := ENTERO$;
$tmp := NuevaTemporal()$;
$T.cod :=$ `mov #`$||$**numentero**$.lexema||tmp$;
$T.dir := tmp$ }

$T \longrightarrow$ **numreal** { $T.tipo := REAL$;
$tmp := NuevaTemporal()$;
$T.cod :=$ `mov $`$||$**numreal**$.lexema||tmp$;
$T.dir := tmp$ }

$T \longrightarrow$ **id** { *if*(($simbolo =$ `tsActual->buscarSimbolo(`**id**$.lexema$`)`$) == null$)
`errorSemantico(...)`
else
$tmp := NuevaTemporal()$;
$T.cod =$ `mov `$||simbolo$`.posicion`$||tmp$;
$T.tipo := simbolo$`.tipo`;
$T.dir := tmp$
endif }

$E \longrightarrow T$ { $E.tipo := T.tipo$; $E.cod := T.cod$; $E.dir := T.dir$ }

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

$E \longrightarrow E$ **opas** T {
$tmp := NuevaTemporal()$; $E.dir := tmp$;
if($E_1.tipo == ENTERO$ `&&` $T.tipo == ENTERO$)
$E.cod := E_1.cod||T.cod||$`mov` $E_1.dir$ `A`$||$
opas$.trad||$`i` $||T.dir||$`mov A` $||tmp$;
$E.tipo := ENTERO$
elsif($E_1.tipo == REAL$ `&&` $T.tipo == ENTERO$)
$tmpcnv := NuevaTemporal()$;
$E.cod := E_1.cod||T.cod||$`mov` $T.dir$ `A`$||$
`itor` $||$`mov A` $tmpcnv||$`mov` $E_1.dir$ `A`$||$
opas$.trad||$`r` $||tmpcnv||$`mov A` $||tmp$;
$E.tipo := REAL$
elsif($E_1.tipo == ENTERO$ `&&` $T.tipo == REAL$)
$E.cod := E_1.cod||T.cod||$`mov` $E_1.dir$ `A`$||$
`itor` $||$**opas**$.trad||$`r` $||T.dir||$`mov A` $||tmp$;
$E.tipo := REAL$
else `// REAL && REAL`
$E.cod := E_1.cod||T.cod||$`mov` $E_1.dir$ `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
`==`	`eqli/eqlr`
`!=`	`neqi/neqr`
`>`	`gtri/gtrr`
`>=`	`geqi/geqr`
`<`	`lssi/lssr`
`<=`	`leqi/leqr`

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 si el lenguaje permite usar solamente dos valores booleanos, como hace Pascal, o si permite usar cualquier valor numérico como valor booleano, como hacen C y C++. El código intermedio 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 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 evalua *A*, se evalua *B*, y se evalua 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 evalua *A*, y solamente si el resultado es *cierto* se evalua *B* (si *A* es *falso* no vale la pena evaluar *B*, el resultado va a ser *falso*)
   - OR : se evalua *A*, y si es *falso* se evalua *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* $\Rightarrow$ `if` *A* `then` *B*

   A `||` *B* $\Rightarrow$ `if` *A* `then` cierto `else` *B*

Generación de código `m2r` para instrucciones (1)

- Asignación:

$$\textit{Instr} \longrightarrow \textbf{id}\ \textbf{asig}\ \textit{Expr}$$

conversiones? (`itor/rtoi`)
`mov` *Expr*.dir **id**.posicion

  - 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

$$\textit{Instr} \longrightarrow \textbf{write}\ \textit{Expr}$$

  - 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

$$\textit{Instr} \longrightarrow \textbf{read}\ \ \textbf{id}$$

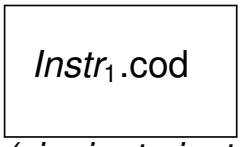
  - Si la variable es de tipo entero:
    `rdi` **id**.posicion
  - Si la variable es de tipo real:
    `rdr` **id**.posicion
  - Si la variable es de tipo carácter (o booleano?):
    `rdc` **id**.posicion

Generación de código `m2r` para instrucciones (4)

- Condicional

$$\textit{Instr} \longrightarrow \textbf{if (}\ \textit{Expr}\ \textbf{)}\ \textit{Instr}_1$$


```
mov  Expr.dir  A      // debe ser un valor entero
jz   L1               // saltar si vale 0 (false)
```



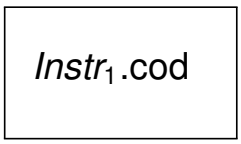

```
L1:  (siguiente instrucción)
```


Generación de código `m2r` para instrucciones (5)

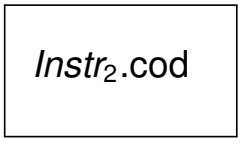
- Condicional (2)

$$\textit{Instr} \longrightarrow \textbf{if (}\ \textit{Expr}\ \textbf{)}\ \textit{Instr}_1\ \textbf{else}\ \textit{Instr}_2$$


```
mov  Expr.dir  A
jz   L1
```




```
jmp  L2
L1:
```




```
L2:  (siguiente instrucción)
```

Generación de código `m2r` para instrucciones (6)

- Iteración

$$\textit{Instr} \longrightarrow \textbf{while (} \textit{Expr} \textbf{)} \ \textit{Instr}_1$$

```
L1:   <Expr.cod>
      mov  Expr.dir  A
      jz   L2
      <Instr1.cod>
      jmp  L1
L2:   (siguiente instrucción)
```


Ejercicio 1

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

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

$$\textit{Instr} \longrightarrow \textbf{for (} \textit{Expr}_1 \textbf{ ; } \textit{Expr}_2 \textbf{ ; } \textit{Expr}_3 \textbf{)} \ \textit{Instr}_1$$

   La expresión $\textit{Expr}_1$ se ejecuta una vez al principio del bucle, la expresión $\textit{Expr}_2$ se ejecuta en cada paso del bucle, y si el resultado es cierto se ejecuta el código de la instrucción, y la expresión $\textit{Expr}_3$ se ejecuta después del código de la instrucción en cada paso del bucle.

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

$$\textit{Instr} \longrightarrow \textbf{do} \ \textit{Instr}_1 \ \textbf{while (} \textit{Expr} \textbf{)}$$

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

3. `repeat-until` de Pascal

$$\textit{Instr} \longrightarrow \textbf{repeat} \ \textit{Instr}_1 \ \textbf{until (} \textit{Expr} \textbf{)}$$

   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

Los tipos de `a`, `b` y `c` en la tabla de símbolos son 3, 5 y 8, respectivamente.

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	TAMAÑO	TIPO BASE	INICIO
1	ENTERO			
2	REAL			
3	ARRAY	10	1	1
4	ARRAY	5	1	21
5	ARRAY	7	4	10
6	ARRAY	35	2	1
7	ARRAY	25	6	1
8	ARRAY	15	7	1

Es necesario almacenar, además del tamaño, el inicio del rango de índices. También se puede guardar el final y calcular el tamaño (tamaño=final-inicio+1).

Tipos compuestos: arrays (4)

ETDS para guardar *arrays* en la tabla de tipos

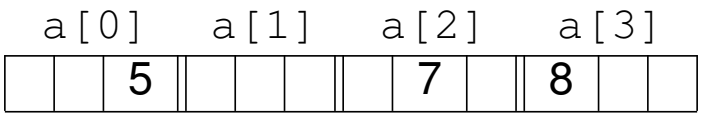
D	$\rightarrow$	T **id** {`tsActual->nuevoSimbolo`(**id**.*lexema*, T.*tipo*, T.*tam*); L.*th* := T.*tipo*; L.*tah* := T.*tam*} L
T	$\rightarrow$	**real** { T.*tipo* := `REAL`; T.*tam* := 1 }
T	$\rightarrow$	**entero** { T.*tipo* := `ENTERO`; T.*tam* := 2 }
T	$\rightarrow$	**tabla num de** T { T.*tipo* := `ttipos->nuevoTipo`(**num**.*lexema*, T_1.*tipo*); T.*tam* := **num**.*valor* $*$ T_1.*tam* }
L	$\rightarrow$	**,** **id** {`tsActual->nuevoSimbolo`(**id**.*lexema*, L.*th*, L.*tah*); L_1.*th* := L.*th*; L_1.*tah* := L.*tah*} L
L	$\rightarrow$	ϵ

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;
```



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]  ...
```

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

- Se suele utilizar una formulación recursiva:

$$\begin{array}{lcll} t_1 & := & 0 & \texttt{// base de la recursión} \\ t_2 & := & t_1 \times 10 + i & \\ t_3 & := & t_2 \times 20 + j & \\ t_4 & := & t_3 \times 30 + k & \\ t_5 & := & \texttt{dir(a)} + t_4 \times \texttt{sizeof(int)} & \texttt{// paso final} \end{array}$$

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]  ...
```

$$\texttt{dir(a[i,j,k])} = \texttt{dir(a)} + (i-1)\times(6\times 11\times \texttt{sizeof(integer)}) + (j-15)\times(11\times \texttt{sizeof(integer)}) + (k-1234)\times \texttt{sizeof(integer)}$$

- Con la formulación recursiva:

$t_1 := 0$ `// base de la recursión`
$t_2 := t_1 \times 10 + i - 1$
$t_3 := t_2 \times 6 + j - 15$
$t_4 := t_3 \times 11 + k - 1234$
$t_5 := \texttt{dir(a)} + t_4 \times \texttt{sizeof(integer)}$ `// 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)

$R \longrightarrow$ **id** { *if*((*simbolo* = `tsActual->buscarSimbolo`(**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* }

$R \longrightarrow R$ **[** { *if*(!`esArray`(R_1.*tipo*))`errorSemantico(...)` }
E **]** { *if*(*E.tipo*! = *ENTERO*)`errorSemantico(...)`
else
R.*tipo* := `ttipos->tipoBase`(R_1.*tipo*);
R.*dbase* := R_1.*dbase*;
tmp := *NuevaTemporal*(); *R.dir* := *tmp*;
R.cod = R_1.*cod*||*E.cod*||
`mov ||`R_1.*dir*|| `A||`
`muli #||ttipos->tamaño(`R_1.*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}
```


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:

$$
\begin{array}{lcl}
I & \longrightarrow & \textbf{print}\ T \\
T & \longrightarrow & T\ \textbf{opas}\ F \\
T & \longrightarrow & F \\
T & \longrightarrow & \textbf{opas}\ F \\
F & \longrightarrow & \textbf{numeroentero} \\
F & \longrightarrow & R \\
F & \longrightarrow & \textbf{(}\ T\ \textbf{)} \\
R & \longrightarrow & \textbf{id}\ D \\
D & \longrightarrow & D\ \textbf{[}\ T\ \textbf{]} \\
D & \longrightarrow & D\ \textbf{\{}\ T\ \textbf{\}} \\
D & \longrightarrow & \epsilon
\end{array}
$$

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:

$$
\begin{array}{lcl}
I & \longrightarrow & \textbf{print}\ T \\
T & \longrightarrow & T\ \textbf{opas}\ F \\
T & \longrightarrow & F \\
T & \longrightarrow & \textbf{opas}\ F \\
F & \longrightarrow & \textbf{numeroentero} \\
F & \longrightarrow & R \\
F & \longrightarrow & \textbf{(}\ T\ \textbf{)} \\
R & \longrightarrow & \textbf{id}\ A \\
A & \longrightarrow & \epsilon \\
A & \longrightarrow & \textbf{[}\ D\ \textbf{]} \\
D & \longrightarrow & D\ \textbf{coma}\ T \\
D & \longrightarrow & T
\end{array}
$$

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.