UNIVERSIDAD
NACIONAL
DE COLOMBIA

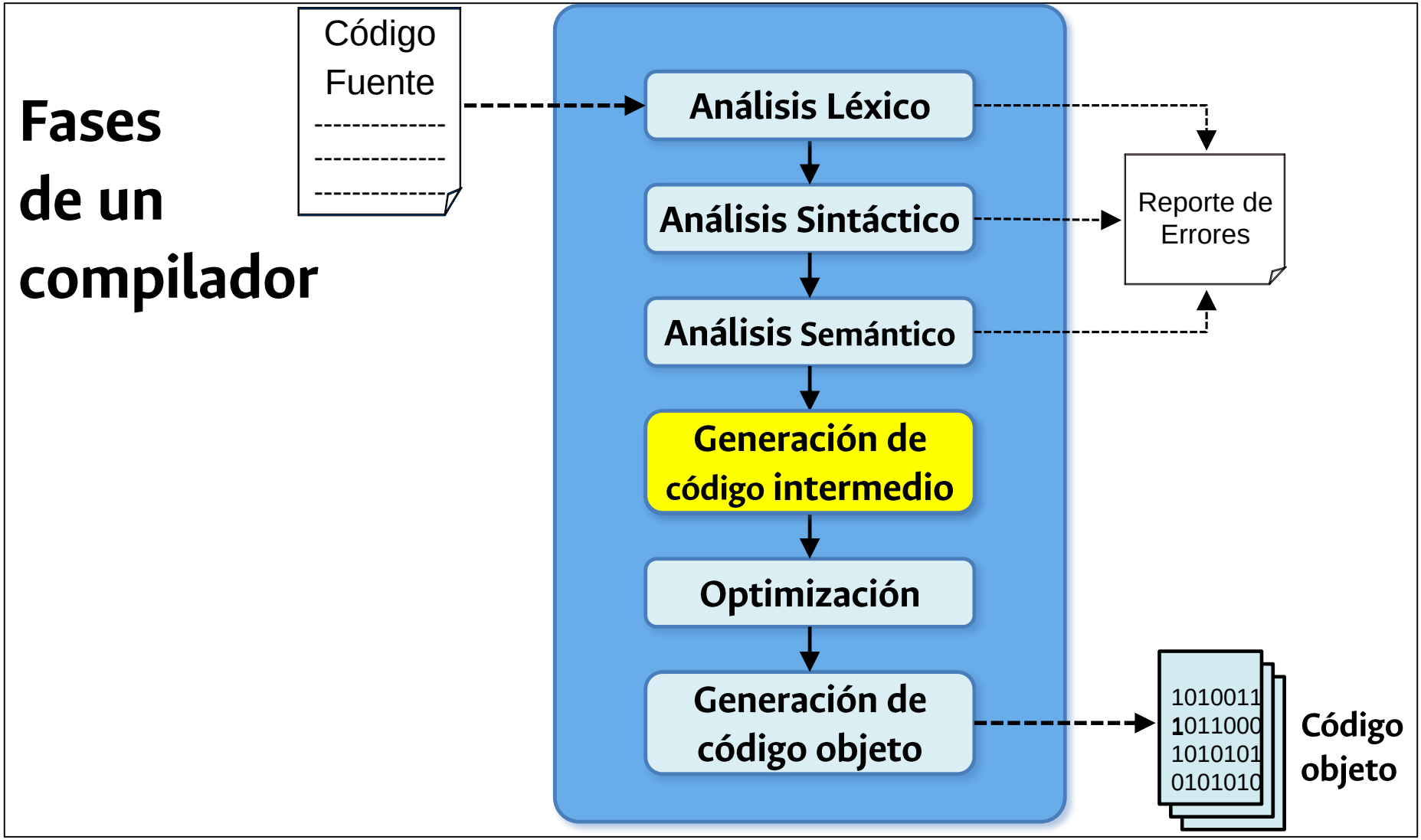
Generación de código intermedio

Felipe Restrepo Calle
ferestrepoca@unal.edu.co

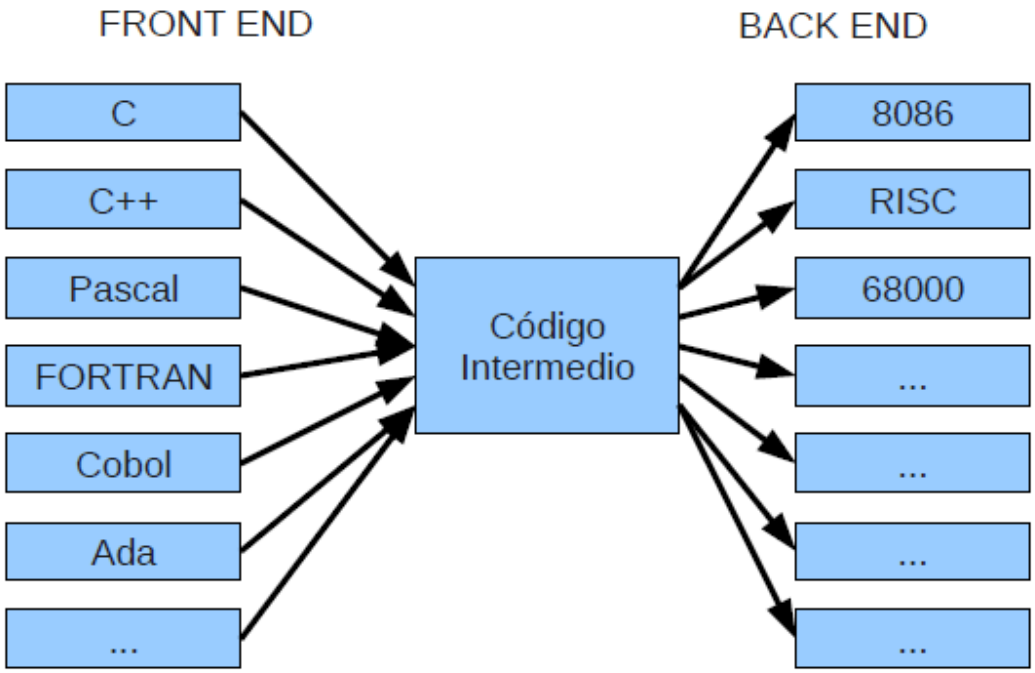
Lenguajes de Programación
Departamento de Ingeniería de Sistemas e Industrial
Facultad de Ingeniería
Universidad Nacional de Colombia
Sede Bogotá

0. Introducción
1. Generación de **código intermedio**
2. Definición de la **tabla de símbolos**
3. Generación de código para **expresiones**
4. Generación de código para **instrucciones**
5. Ejercicios

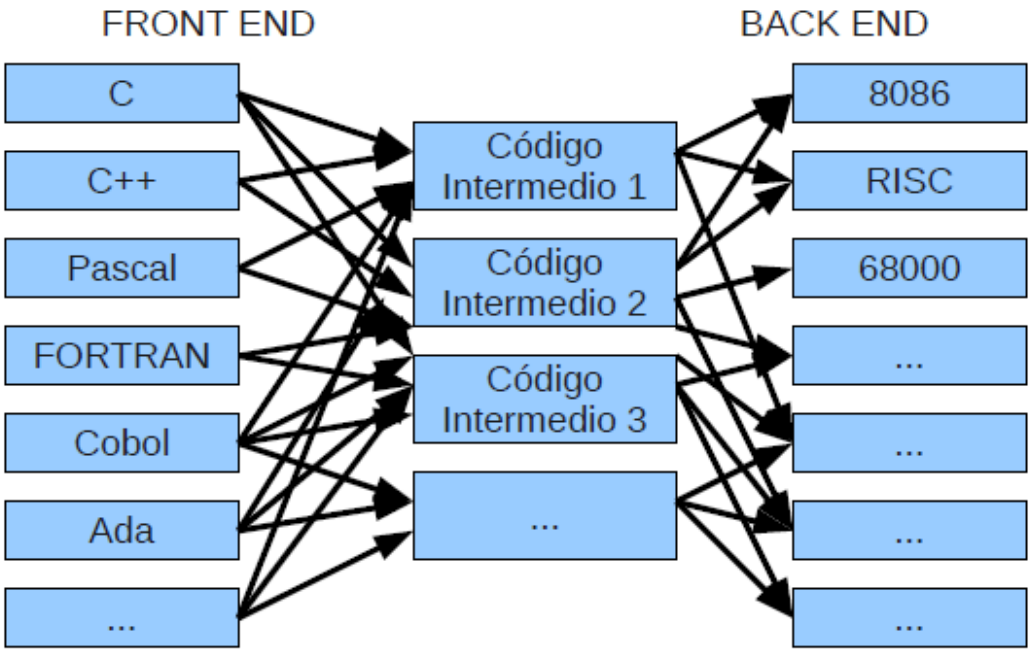
Fases de un compilador



Código en una representación intermedia independiente de la sintaxis del lenguaje y de la arquitectura

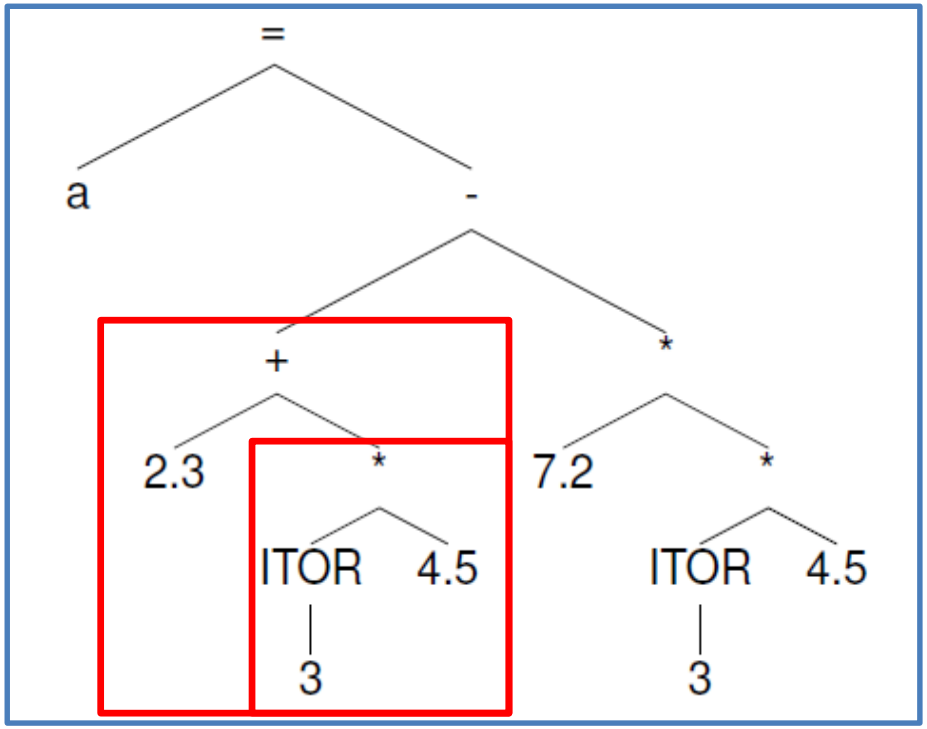


Código en una representación intermedia independiente de la sintaxis del lenguaje y de la arquitectura



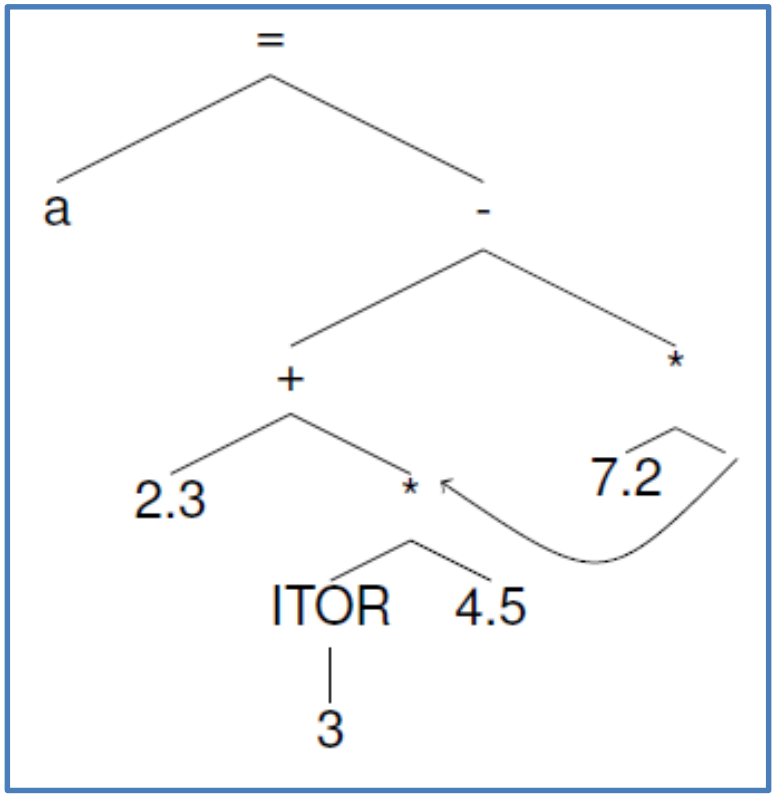
Árbol de sintaxis abstracta (AST)

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



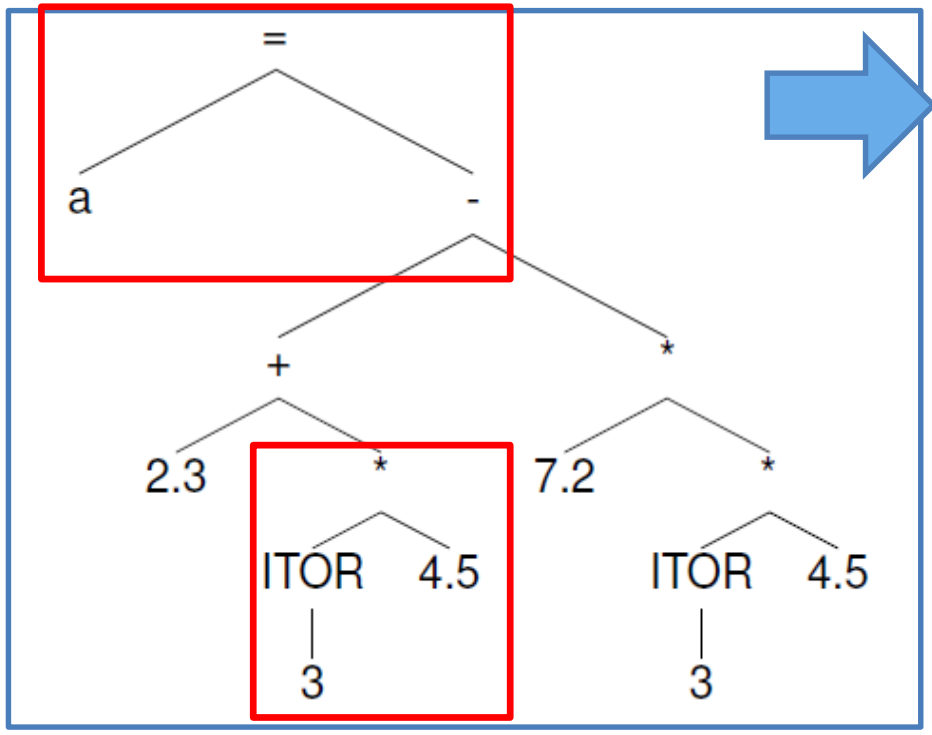
Grafo dirigido acíclico (**DAG** – Directed acyclic graph)

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



Código de tres direcciones (3AC – Three address code)

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



Código de tres direcciones (3AC)

```
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
```

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

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

```
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
...
```

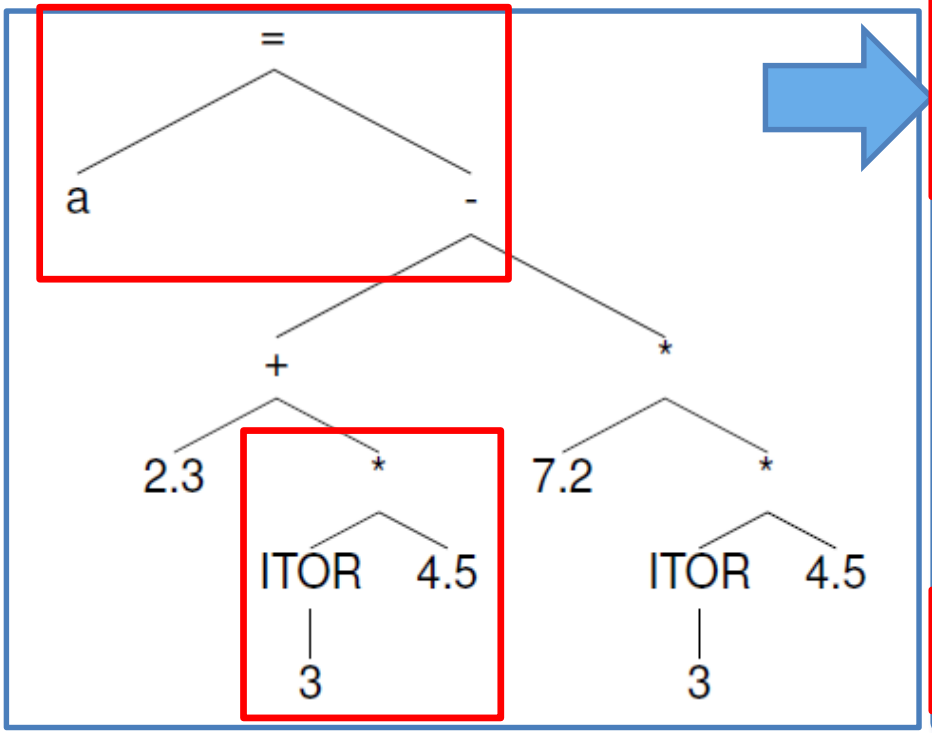
Máquina virtual de pila (como P-code, usado en los primeros compiladores de Pascal)

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

```
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
```

Máquina virtual de pila (Common Inte - **CIL** - de .NET y Mono)

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



Máquina virtual de pila (CIL de.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 número de parámetros
- etiqueta de comienzo del código, ...
- tipos definidos por el usuario: nombre, tipo, tamaño, ...

Tabla de símbolos

En la tabla de símbolos se almacena información de cada símbolo que aparece en el programa fuente:

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

 ...
}
```

```
.method static public void main() cil managed
{
  .locals (int32, int32, float64, float64)
  .entrypoint
  .maxstack ...

  ...
}
```

Tabla de símbolos

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
- `getTipo`, `getPosicion`, ...: obtener información del símbolo de la tabla de símbolos

Tabla de símbolos

Implementación:

- Se suele utilizar una **tabla hash**. Es muy eficiente para el almacenamiento de identificadores.
- La función de búsqueda `buscarSimbolo` retorna toda la información del símbolo (un objeto de la clase *Simbolo*), de manera que el acceso a cada atributo del símbolo no implicará una nueva búsqueda en la tabla.

Tabla de símbolos

Al gestionar la tabla se deben tener en cuenta los **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
}
```

- Se puede abrir un nuevo ámbito en el que es posible declarar símbolos con el mismo nombre que en ámbitos anteriores
- Cuando se acaba el bloque, se deben olvidar las variables declaradas en ese ámbito

Tabla de símbolos

Implementación de la tabla de símbolos con ámbitos anidados:

1. Usar un **arreglo 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 hasta el principio del arreglo (`buscarSimbolo`)

2. Usar una **pila de tablas de símbolos**:
   - Cada tabla de símbolos almacena 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 anteriores

ETDS para gestionar la tabla de símbolos

D	→	*T* **id** {`tsActual.nuevoSimbolo`(**id**.*lexema*, *T.tipo*); *L.th* := *T.tipo*} *L*
T	→	**float** {*T.tipo* := *REAL*}
T	→	**int** {*T.tipo* := *ENTERO*}
L	→	**,** **id** {`tsActual.nuevoSimbolo`(**id**.*lexema*, *L.th*); L_1.*th* := *L.th*} *L*
L	→	ϵ
...		
Instr	→	**id** {*if*((*simbolo* = `tsActual.buscarSimbolo`(**id**.*lexema*) == *null*) `errorSemantico(...)`}
		asig *Expr* { ... *Instr.trad* := *Expr.trad*|| ... ||`stloc` *simbolo*.`getPosicion()` }
...		
Factor	→	**id** {*if*((*simbolo* = `tsActual.buscarSimbolo`(**id**.*lexema*) == *null*) `errorSemantico(...)` *else* *Factor.trad* := `ldloc` *simbolo*.`getPosicion()` *Factor.tipo* := *simbolo*.`getTipo()` *endif*}

ETDS para gestionar los ámbitos

```
S      →  {tsActual = new TablaSimbolos(null)} SecSp
...
Sp     →  TipoFuncion id (
            {tsActual.nuevoSimbolo(id.lexema, TipoFuncion.tipo)
             tsActual = new TablaSimbolos(tsActual)}
          Args ) Bloque
            { ...
              tsActual = tsActual.pop()
              ...
            }
...
Instr  →          { tsActual = new TablaSimbolos(tsActual)}
          Bloque  { ...
                    tsActual = tsActual.pop()
                    ...
                  }
```

Gestión de ámbitos en CIL:

Todas las variables locales se deben poner al principio del código del método

```
int func()                .method int func() cil managed
{                         {
  double a,b;               .locals (float64, float64, int32, int32)
  {                         .maxstack ...
    int c,a;              }
  }
}
```

Se les puede poner nombre a las variables en CIL:

```
.locals (float64 'a', float64 'b', int32 'c', int32 'a')
```

pero hay que tener en cuenta los ámbitos:

```
.locals (float64 'a_0', float64 'b_0', int32 'c_1', int32
'a_1')
```

Sistema de tipos

Cada lenguaje 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 *warnings*. Ej:

```
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).

Sistema de tipos

Ejemplo:

Javascript

```
var i = 1;
i = i + "";
i + 1;  // evaluates to the String '11'
i - 1;  // evaluates to the Number 0
```

ETDS para el cálculo de tipos en expresiones

$E \longrightarrow E$ **opsum** 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*.getTipo()
  endif }

ETDS para el cálculo de tipos en expresiones (Otra forma)

```
E  ⟶  E opsum T {
          if(E1.tipo == ENTERO && T.tipo == ENTERO)
             E.tipo := ENTERO
          else // cualquier otra combinación
             E.tipo := REAL
          }
E  ⟶  T {E.tipo := T.tipo}
T  ⟶  numentero {T.tipo := ENTERO}
T  ⟶  numreal {T.tipo := REAL}
T  ⟶  id {if((simbolo = tsActual.buscarSimbolo(id.lexema) == null)
           errorSemantico(...)
         else
           ...
           T.tipo := simbolo.getTipo()
         endif }
```

Generación de código para expresiones en CIL

- En CIL se utiliza (virtualmente) una pila para realizar las operaciones.
- Operaciones para apilar valores:
  - `ldc.i4 numentero` : apila un número entero
  - `ldc.r8 numreal` : apila un número real
  - `ldloc posicion` : apila el valor de la variable local en esa posición
  - `ldarg posicion` : apila el valor del argumento de esa posición
  - …
- Operaciones aritméticas (sobrecargadas): `add, sub, mul, div,…`
- Operaciones de conversión: `conv.r8, conv.i4`
- La directiva `.maxstack` debe indicar el tamaño máximo de pila que usa cada método. El compilador debe calcular el tamaño máximo de pila de cada expresión.

Generación de código para expresiones en CIL (ETDS)

$T \longrightarrow$ **numentero** { *T.tipo* := *ENTERO*;
T.cod := `ldc.i4` ||**numentero**.*lexema*;
T.maxstack := 1 }

$T \longrightarrow$ **numreal** { *T.tipo* := *REAL*;
T.cod := `ldc.r8` ||**numreal**.*lexema*;
T.maxstack := 1 }

$T \longrightarrow$ **id** { *if* ((*simbolo* = `tsActual.buscarSimbolo`(**id**.*lexema*) == *null*)
`errorSemantico(...)`
else
T.cod = `ldloc` ||*simbolo*.`getPosicion()`
T.tipo := *simbolo*.`getTipo()`
T.maxstack := 1
endif }

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

Generación de código para expresiones en CIL (ETDS)

$E \longrightarrow E$ **opsum** T {

if(E_1.*tipo* == *ENTERO* && *T.tipo* == *ENTERO*)

E.tipo := *ENTERO*

E.cod := E_1.*cod*||*T.cod*||**opsum**.*trad*

elsif(E_1.*tipo* == *REAL* && *T.tipo* == *ENTERO*)

E.tipo := *REAL*

E.cod := E_1.*cod*||*T.cod*||`conv.r8`||**opsum**.*trad*

elsif(E_1.*tipo* == *ENTERO* && *T.tipo* == *REAL*)

E.tipo := *REAL*

E.cod := E_1.*cod*||`conv.r8`||*T.cod*||**opsum**.*trad*

else `// REAL && REAL`

E.tipo := *REAL*

E.cod := E_1.*cod*||*T.cod*||**opsum**.*trad*

endif

E.maxstack := `MAX(`E_1.*maxstack*, 1 + *T.maxstack*`)`

}

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

Operadores relacionales en CIL

Las instrucciones en CIL para los operadores relacionales son:

- `ceq` desapila dos valores, primero v2 y después v1, y deja un 1 en la pila si v1 == v2, y un 0 si no lo son
- `cgt` desapila dos valores, y deja un 1 si v1 > v2
- `clt` desapila dos valores, y deja un 1 si v1 < v2

Para los operadores restantes es necesario usar una de estas instrucciones y luego negar lógicamente el resultado, ejemplo:
v1 >= v2 → not(v1 < v2):

```
... // v1 y v2 están en la pila
clt // v1<v2
ldc.i4 0
ceq // negar es equivalente a ver si es igual a 0
```

Operadores booleanos

- Los operadores booleanos sirven para *verdadero* y *falso*. Lenguajes como C/C++ asumen que un 0 es *falso* y cualquier valor distinto de 0 es *verdadero*, 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 (Pascal), o si permite usar cualquier valor numérico (C/C++). El código intermedio que se debe generar en ambos casos será diferente, dependiendo de las instrucciones del lenguaje intermedio.
- Es recomendable que los valores booleanos se representen internamente en el código intermedio con los valores 0 y 1.

Operadores booleanos

Ejemplo: **Javascript:** expresiones que se evalúan como **true**

```
"1" + "2" == "12"
"1" - 2 == -1
"2" * "3" == 6
(x="1", ++x) == 2
0 == -0
1/0 != 1/-0
[123] == 123
[123][0] != 123[0]
[0] == false
([0] ? true : false) == true
(a=[0], a==a && a==!a) == true
[] == ![]
" \t\r\n" == 0
",,," == Array((null,'cool',false,NaN,4))
new Array([],null,undefined,null) == ",,,"
```

```
Array() == false
'' != '0'
0 == ''
0 == '0'
false != 'false'
false == '0'
false != undefined
false != null
null == undefined
NaN != NaN
[] + [] == ""
[] + {} == "[object Object]"
{} + [] == 0
(typeof NaN) == "number"
("S"-1=="S"-1) == false
```

Operadores booleanos

Dada una expresión **`A op B`**, para evaluar los operadores AND y OR:

1. **Evaluación similar a la de otros operadores binarios**. En Pascal se evalúa A (que deja su valor en la pila), se evalúa B, y se evalúa la operación AND u OR.

   En CIL se debe utilizar el hecho de que el operador AND es equivalente al producto de booleanos y el operador OR es equivalente a la suma.

2. **Evaluación en cortocircuito:**

   AND: se evalúa A, y solamente si el resultado es verdadero se evalúa B
   OR: se evalúa A, y si es falso se evalúa B

   La implementación se realiza con saltos condicionales:

   `A && B` → `if A then B`
   `A || B` → `if A then` *`verdadero`* `else B`

Generación de código CIL para instrucciones

1. Asignación:

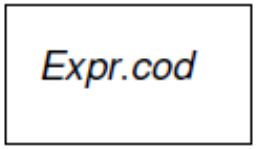
Instr ⟶ **id asig** *Expr*

- El código generado para las expresiones deja el valor de la expresión en la cima de la pila.
- Dependiendo del lenguaje, puede ser necesario hacer conversiones entre tipos o producir errores semánticos.
- La orden `stloc` es para variables locales de un método/función, habría que usar otras órdenes similares para argumentos, campos, atributos, etc.

Generación de código CIL para instrucciones

2. Salida

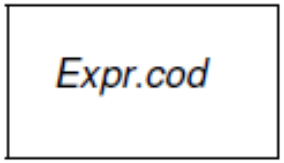
Instr ⟶ **write** *Expr*




```
call void [mscorlib]System.Console::Write(int32)
```

- Si la expresión es de tipo real, habría que usar `Write(float64)`
- Si queremos escribir un real con formato, por ejemplo con 8 cifras y 3 decimales, habría que poner:

```
ldstr "0,8:F3"
```




```
box [mscorlib]System.Double
call void [mscorlib]System.Console::Write(string,object)
```

Generación de código CIL para instrucciones

3. Entrada

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

- Si la variable es de tipo entero:

```
call string [mscorlib]System.Console::ReadLine()
call int32 [mscorlib]System.Int32::Parse(string)
stloc id.posicion
```

- Si la variable es de tipo real:

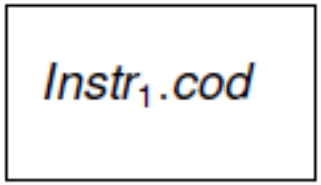
```
call string [mscorlib]System.Console::ReadLine()
call float64 [mscorlib]System.Double::Parse(string)
stloc id.posicion
```

Generación de código CIL para instrucciones

4. Condicional if

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


```
ldc.i4 0 // o ldc.r8 0.0 si la expresión es real
beq L1 // saltar si la expresión vale 0 (false)
```

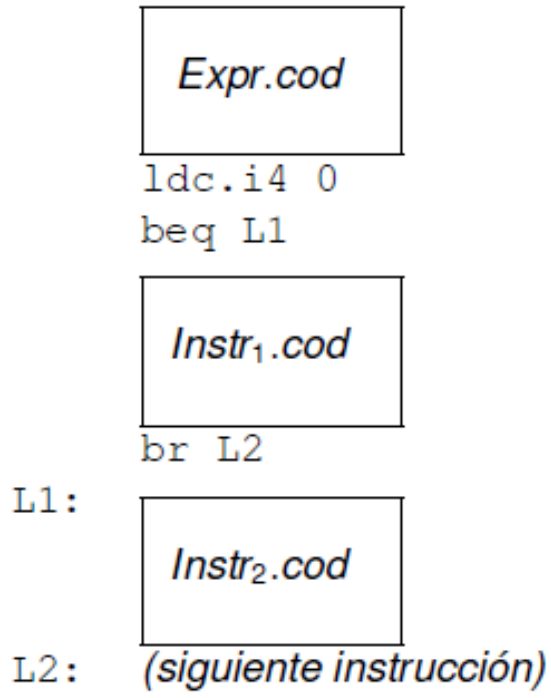



```
L1:
```
(siguiente instrucción)

Generación de código CIL para instrucciones

5. Condicional if-then-else

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



Generación de código CIL para instrucciones

6. Ciclo while

Instr ⟶ **while** (*Expr*) $Instr_1$

```
L1:   [ Expr.cod ]
      ldc.i4 0
      beq L2
      [ Instr1.cod ]
      br L1
L2:   (siguiente instrucción)
```

Ejercicios

Indicar qué código se genera en CIL para las siguientes instrucciones:

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

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

La expresión *Expr1* se ejecuta una vez al principio del ciclo; la expresión *Expr2* se ejecuta en cada paso del ciclo, y si su resultado es *verdadero* se ejecuta el código de *Instr1*; la expresión *Expr3* se ejecuta después del código de la instrucción en cada paso del ciclo.

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

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

La instrucción se ejecuta al menos una vez y se repite mientras la expresión sea *Verdadera*.

3. **`repeat-until`** de Pascal

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

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

Ejercicios

4. Considerar el siguiente fragmento de la especificación sintáctica de un lenguaje:

$$
\begin{array}{lcl}
S & \longrightarrow & L \\
L & \longrightarrow & L\ I \\
L & \longrightarrow & I \\
I & \longrightarrow & \textbf{break} \\
I & \longrightarrow & \textbf{print entero} \\
I & \longrightarrow & \textbf{switch id llavei}\ T\ \textbf{llaved} \\
T & \longrightarrow & C\ T \\
T & \longrightarrow & D \\
C & \longrightarrow & \textbf{case entero dosp}\ P \\
D & \longrightarrow & \textbf{default dosp}\ P \\
P & \longrightarrow & L \\
P & \longrightarrow & \epsilon
\end{array}
$$

Diseñar un esquema de traducción dirigida por la sintaxis (ETDS) que genere el código adecuado para CIL y que emita los mensajes de error semántico oportunos.
(continúa)

Ejercicios

4. (Continuación ejercicio 4)

Consideraciones:

- El único tipo de dato simple de este lenguaje es el tipo entero.
- Esta parte de la gramática caracteriza principalmente la instrucción de selección, que funciona a la manera de C/C++. Considerar, por ejemplo, un programa como el siguiente:

```
switch a {
        case 1:
        case 5:  print 100
        case 2:  print 200
                 break
        case 3:  print 300
                 break
        default:
}
print 901
```

Si la variable *a* vale 1 o 5 se imprime 100, 200 y 901; si vale 2, se imprime 200 y 901 (la instrucción `break` hace que la ejecución siga tras el `switch`); si la variable *a* vale 3, se imprimirá 300 y 901; finalmente, si tiene cualquier otro valor, se imprime 901 *(continúa)*

Ejercicios

4. (Continuación ejercicio 4)

Consideraciones:

- Asumir que la tabla de símbolos actual es accesible a través de la referencia global TSA
- La instrucción break no puede aparecer fuera de un `switch`; si lo hace, habrá que emitir el mensaje de error
- Asumir que:
  - los identificadores son siempre variables locales
  - el ETDS obtiene en un atributo sintetizado del símbolo inicial ***S*** el tamaño mínimo de la pila (necesario para la directiva `maxstack`) para que el código generado se ejecute correctamente en la máquina virtual
  - la signatura de la función para imprimir un entero es:

```
void [mscorlib]System.Console::Write(int32)
```