Tema 5: Traducción dirigida por la sintaxis

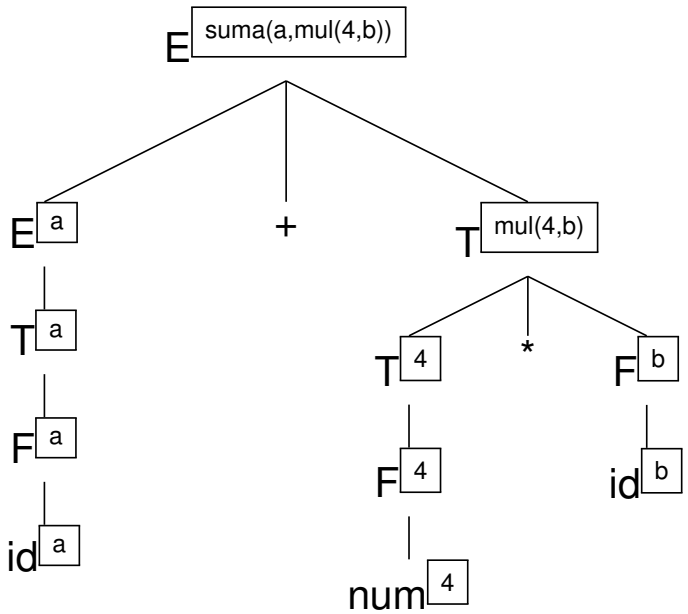
Procesamiento de Lenguajes

Dept. de Lenguajes y Sistemas Informáticos
Universidad de Alicante

Ejemplo 1 de traducción

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & \textbf{id} \\
F & \rightarrow & \textbf{num}
\end{array}
$$

```
a+4*b
  ⇓
suma(a,mul(4,b))
```



Ejemplo 1 de traducción (2)

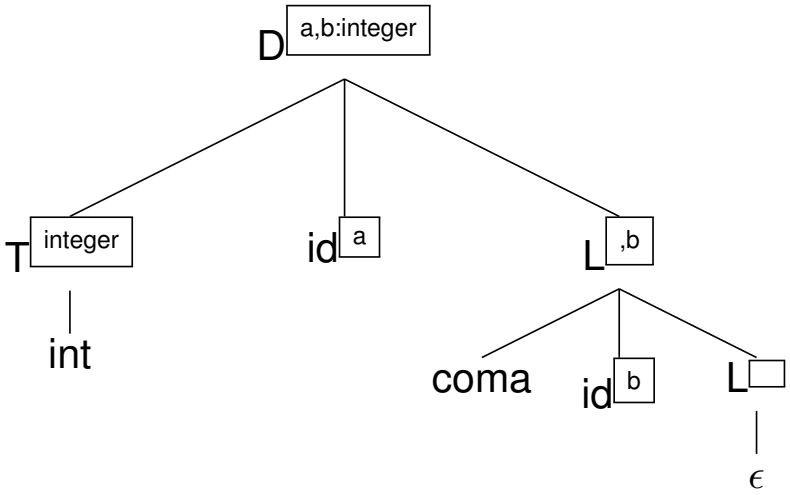
GRAMÁTICA DE ATRIBUTOS

REGLA	ACCIÓN SEMÁNTICA
$E \rightarrow E$ **+** T	$E.trad := "suma(" \|\| E_1.trad \|\| "," \|\| T.trad \|\| ")"$
$E \rightarrow T$	$E.trad := T.trad$
$T \rightarrow T$ * F	$T.trad := "mul(" \|\| T_1.trad \|\| "," \|\| F.trad \|\| ")"$
$T \rightarrow F$	$T.trad := F.trad$
$F \rightarrow$ **id**	$F.trad :=$ **id**$.lexema$
$F \rightarrow$ **num**	$F.trad :=$ **num**$.lexema$

IMPORTANTE: no es posible acceder a atributos de símbolos que no estén en la regla

Ejemplo 2 de traducción

$D \rightarrow T$ **id** L
$T \rightarrow$ **float**
$T \rightarrow$ **int**
$L \rightarrow$ **coma id** L
$L \rightarrow \epsilon$




```
int a,b
   ⇓
a,b:integer
```

Ejemplo 2 de traducción (2)

GRAMÁTICA DE ATRIBUTOS	
REGLA	ACCIÓN SEMÁNTICA
$D \rightarrow T$ **id** L	$D.trad := \textbf{id}.lexema \| \| L.trad \| \| " : " \| \| T.trad$
$T \rightarrow$ **float**	$T.trad := "real"$
$T \rightarrow$ **int**	$T.trad := "integer"$
$L \rightarrow$ **coma id** L	$L.trad := " , " \| \| \textbf{id}.lexema \| \| L_1.trad$
$L \rightarrow \epsilon$	$L.trad := ""$

Implementación del traductor

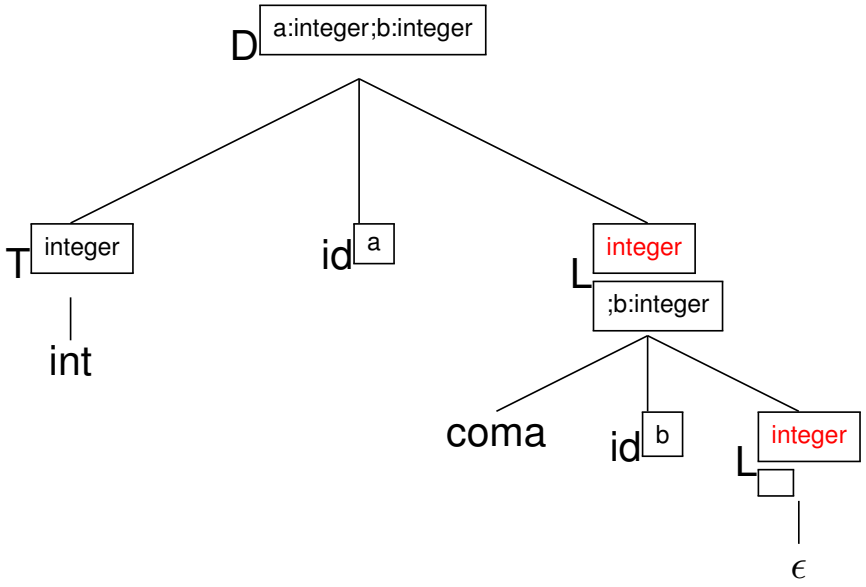
Existen dos posibilidades:

1. El analizador sintáctico construye el árbol (decorado con los atributos de los terminales), y en una segunda pasada se recorre el árbol calculando los atributos que falten hasta completar la traducción (traducción de dos o más pasadas)
2. El analizador sintáctico no construye explícitamente el árbol (aunque hace un recorrido virtual por el árbol), y va calculando todos los atributos a la vez que va recorriendo el árbol (traducción de una sola pasada)

Ejemplo 3 de traducción

$D \rightarrow T\ \textbf{id}\ L$
$T \rightarrow \textbf{float}$
$T \rightarrow \textbf{int}$
$L \rightarrow \textbf{coma}\ \textbf{id}\ L$
$L \rightarrow \epsilon$

```
int a,b
   ⇓
a:integer;b:integer
```



Ejemplo 3 de traducción (2)

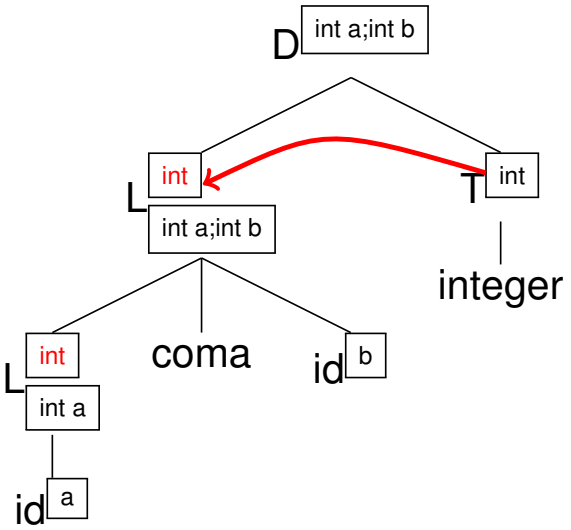
GRAMÁTICA DE ATRIBUTOS	
REGLA	ACCIÓN SEMÁNTICA
$D \rightarrow T\ \textbf{id}\ L$	$L.th := T.trad$; $D.trad := \textbf{id}.lexema \|\| " : " \|\| T.trad \|\| L.trad$
$T \rightarrow \textbf{float}$	$T.trad := "real"$
$T \rightarrow \textbf{int}$	$T.trad := "integer"$
$L \rightarrow \textbf{coma}\ \textbf{id}\ L$	$L_1.th := L.th$; $L.trad := ";" \|\| \textbf{id}.lexema \|\| " : " \|\| L.th \|\| L_1.trad$
$L \rightarrow \epsilon$	$L.trad := ""$

- Algunos no terminales tienen más de un atributo
- Algunos atributos se *calculan* a partir de atributos de hermanos o padres en el árbol. Estos atributos se llaman atributos heredados (los demás atributos se llaman sintetizados).

Ejemplo 4 de traducción

$D \rightarrow L\ T$
$T \rightarrow$ **real**
$T \rightarrow$ **integer**
$L \rightarrow L$ **coma id**
$L \rightarrow$ **id**

```
a,b integer
     ⇓
int a;int b
```



IMPORTANTE: hay herencia de derecha a izquierda en el árbol

Ejemplo 4 de traducción (2)

GRAMÁTICA DE ATRIBUTOS	
REGLA	ACCIÓN SEMÁNTICA
$D \rightarrow L\ T$	$L.th := T.trad$; $D.trad := L.trad$
$T \rightarrow$ **real**	$T.trad :=$ "*double*"
$T \rightarrow$ **integer**	$T.trad :=$ "*int*"
$L \rightarrow L$ **coma id**	$L_1.th := L.th$; $L.trad := L_1.trad \|\| ";" \|\| L.th \|\| "\ " \|\|$ **id**.*lexema*
$L \rightarrow$ **id**	$L.trad := L.th \|\| "\ " \|\|$ **id**.*lexema*

- La herencia de derecha (T) a izquierda (L) implica que cuando se está analizando/traduciendo L todavía no se conoce la traducción de T
- Por tanto, no es posible traducir a la vez que se analiza, no es posible la *traducción en una pasada*.

Gramática de atributos por la izquierda

DEFINICIÓN: una gramática de atributos se dice que es una gramática de atributos por la izquierda (*l-attributed grammar*) si solamente hay herencia de padres a hijos o bien de izquierda a derecha en el árbol, pero no de derecha a izquierda
Las gramáticas de atributos por la izquierda (GAI) son aquellas que permiten evaluar los atributos a la vez que se hace el análisis sintáctico (en una sola pasada).

Análisis semántico: tabla de símbolos

La tabla de símbolos se utiliza para almacenar los símbolos (identificadores) declarados en el programa fuente, junto con su tipo y posiblemente alguna información más (dirección de memoria, etc).
Ejemplo:

```
int a,b;
float c,d;
```

NOMBRE	TIPO	DIRECCIÓN
a	ENTERO	0
b	ENTERO	2
c	REAL	4
d	REAL	8

Tabla de símbolos (2)

- Cuando el compilador procesa las declaraciones, tiene que añadir las variables declaradas a la tabla de símbolos (comprobando que no se declara dos veces el mismo identificador en el mismo ámbito).
- En el código (instrucciones, expresiones, etc), cuando aparece un identificador el compilador debe buscarlo en la tabla de símbolos, y obtener sus datos: tipo, dirección, etc.

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 en C/C++ se pueden declarar símbolos con el mismo nombre que en los ámbitos anteriores abiertos , pero en Java no (nunca con el mismo nombre que otros símbolos del mismo ámbito)
- Cuando se acaba el bloque, se deben *olvidar* las variables declaradas en ese ámbito

Implementación de la tabla de símbolos

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

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

Implementación:

- Se suele utilizar una tabla hash, es muy eficiente para el almacenamiento de identificadores
- Con ámbitos anidados:
  1. Usar un vector de símbolos, marcando y guardando el comienzo de cada ámbito, de forma que las operaciones `nuevoSimb` y `buscar` empiecen la búsqueda por el final, y paren al principio del ámbito (`nuevoSimb`) o sigan hacia el principio del vector (`buscar`)
  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 `buscar`, si no se encuentra un símbolo en la tabla actual, se busca recursivamente en las tablas de los ámbitos abiertos anteriores.

Ejemplo 5 de traducción

En este ejemplo, el traductor no genera traducción, pero debe almacenar los símbolos en la tabla de símbolos:

```
int a,b,c
```

NOMBRE	TIPO	DIRECCIÓN
a	ENTERO	0
b	ENTERO	2
c	ENTERO	4

DEFINICIÓN DIRIGIDA POR LA SINTAXIS

REGLA	ACCIÓN SEMÁNTICA
$D \rightarrow T$ **id** L	$L.th := T.tipo$; `nuevoSimb`(**id**.*lexema*, $T.tipo$)
$T \rightarrow$ **float**	$T.tipo := REAL$
$T \rightarrow$ **int**	$T.tipo := ENTERO$
$L \rightarrow$ **coma id** L	$L_1.th := L.th$; `nuevoSimb`(**id**.*lexema*, $L.th$)
$L \rightarrow \epsilon$	

Cuando una gramática de atributos tiene acciones que no calculan atributos, se denomina Definición dirigida por la sintaxis (DDS)

Definiciones dirigidas por la sintaxis (DDS)

- Las DDS son como las gramáticas de atributos, pero pueden incluir acciones que no calculen atributos, con efectos secundarios (como guardar símbolos en la tabla de símbolos), por lo que todas las gramáticas de atributos son también DDS.
- IMPORTANTE: las DDS solamente especifican las acciones que es necesario realizar en cada regla, pero no el orden en el que hay que ejecutarlas. Las DDS son una herramienta para el diseño de alto nivel del traductor, sin entrar en los detalles de implementación. Sin embargo, el orden de ejecución de las acciones es muy importante

Ejemplo 5 de traducción (2)

`int a,b,c`

NOMBRE	TIPO	DIRECCIÓN
a	ENTERO	0
b	ENTERO	2
c	ENTERO	4

ESQUEMA DE TRADUCCIÓN DIRIGIDO POR LA SINTAXIS (ETDS)

D	$\rightarrow$	T **id** {`nuevoSimb`(**id**.*lexema*, T.*tipo*); L.*th* := T.*tipo*} L
T	$\rightarrow$	**float** { T.*tipo* := *REAL*}
T	$\rightarrow$	**int** { T.*tipo* := *ENTERO*}
L	$\rightarrow$	**coma** **id** {`nuevoSimb`(**id**.*lexema*, L.*th*); L_1.*th* := L.*th*} L
L	$\rightarrow$	ϵ

Un esquema de traducción dirigido por la sintaxis (ETDS) es como una DDS en la que las acciones semánticas se insertan en la parte derecha de la regla, en el momento exacto del análisis en el que se tienen que ejecutar.

Esquema de traducción dirigido por la sintaxis (ETDS)

- Los ETDS son la herramienta para diseñar traductores de una sola pasada
- Las acciones semánticas se situan (encerradas entre llaves) en el punto de la parte derecha de la regla en que se deben ejecutar. Se tienen que cumplir las siguientes restricciones:
  1. Un atributo heredado de un símbolo α_i de la parte derecha de la regla se debe calcular en una acción semántica situada antes de α_i
  2. Una acción semántica no puede referirse a un atributo sintetizado de un símbolo situado a la derecha de la acción en la regla
  3. Un atributo sintetizado de la parte izquierda de la regla A solo se puede calcular después de haber calculado todos los atributos que se usan para calcularlo (preferentemente al final de la regla).

Ejemplo 3 de traducción, con ETDS

GRAMÁTICA DE ATRIBUTOS	
REGLA	ACCIÓN SEMÁNTICA
$D \rightarrow T$ **id** L	$L.th := T.trad; D.trad :=$ **id**$.lexema \|\| " : " \|\| T.trad \|\| L.trad$
$T \rightarrow$ **float**	$T.trad :=$ "*real*"
$T \rightarrow$ **int**	$T.trad :=$ "*integer*"
$L \rightarrow$ **coma id** L	$L_1.th := L.th; L.trad :=$ ";" $\|\|$ **id**$.lexema \|\|$ " : " $\|\| L.th \|\| L_1.trad$
$L \rightarrow \epsilon$	$L.trad :=$ ""

ESQUEMA DE TRADUCCIÓN DIRIGIDO POR LA SINTAXIS		
D	$\rightarrow$	T **id** $\{L.th := T.trad\}\, L\ \{D.trad :=$ **id**$.lexema \|\|$ " : " $\|\| T.trad \|\| L.trad\}$
T	$\rightarrow$	**float** $\{T.trad :=$ "*real*" $\}$
T	$\rightarrow$	**int** $\{T.trad :=$ "*integer*" $\}$
L	$\rightarrow$	**coma id** $\{L_1.th := L.th\}\, L\ \{L.trad :=$ ";" $\|\|$ **id**$.lexema \|\|$ " : " $\|\| L.th \|\| L_1.trad\}$
L	$\rightarrow$	$\epsilon\ \{L.trad :=$ "" $\}$

¿Cómo se hace un ETDS?

Información que se necesita:

- Gramática
- Cadenas del lenguaje fuente y su traducción (ejemplos de traducción)
- Otras restricciones (semánticas, . . .)

¿Cómo hacer el ETDS?

1. Construir árboles (o subárboles) para alguna cadena de entrada y estudiar cómo debe generarse la traducción
2. Diseñar las acciones de traducción (acciones semánticas) asociadas a cada regla de la gramática, y elegir la posición de la acción en la parte derecha
3. Comprobar que el ETDS funciona con los ejemplos de traducción

Ejercicio 0 (sin atributos heredados)

Diseña un ETDS para traducir declaraciones sencillas de parámetros en C/C++ a Pascal. Ejemplos:

```
int a                          a:integer
int a,float b                  a:integer;b:real
float &a,int b,int &c          var a:real;b:integer;var c:integer
```

La gramática que debes utilizar para diseñar el ETDS es la siguiente:

$$
\begin{array}{lcl}
L & \longrightarrow & L \ \textbf{,} \ Par \\
L & \longrightarrow & Par \\
Par & \longrightarrow & Tipo \ \textbf{id} \\
Par & \longrightarrow & Tipo \ \textbf{\&} \ \textbf{id} \\
Tipo & \longrightarrow & \textbf{int} \\
Tipo & \longrightarrow & \textbf{float}
\end{array}
$$

Ejercicio 1 (sin atributos heredados)

Diseña un ETDS para traducir declaraciones de vectores y matrices en C/C++ a declaraciones con inicialización de Java. Ejemplo:

```
int matriz[10][20][30];          int[][][]   matriz = new int[10][20][30];
float bidim[15][24];             float[][]   bidim  = new float[15][24];
bool badam[14];                  boolean[]   badam  = new boolean[14];
```

La gramática que debes utilizar para diseñar el ETDS es la siguiente:

S	$\longrightarrow$	*Tipo* **id** *Dim* **pyc**
Dim	$\longrightarrow$	*Dim* **cori entero cord**
Dim	$\longrightarrow$	**cori entero cord**
Tipo	$\longrightarrow$	**int**
Tipo	$\longrightarrow$	**float**
Tipo	$\longrightarrow$	**bool**

Tabla de símbolos (ETDS `nuevoSimb/buscar`)

ETDS para gestionar la tabla de símbolos

D	$\rightarrow$	*T* **id** {`tsActual->nuevoSimb`(**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->nuevoSimb`(**id**.*lexema*, *L.th*); L_1.*th* := *L.th*} *L*
L	$\rightarrow$	ϵ
...		
Instr	$\rightarrow$	**id** {*if*((*simbolo* = `tsActual->buscar`(**id**.*lexema*)) == *null*) `errorSemantico`(...)} **asig** *Expr* { ... *Instr.trad* := ... }
...		
Factor	$\rightarrow$	**id** {*if*((*simbolo* = `tsActual->buscar`(**id**.*lexema*)) == *null*) `errorSemantico`(...) *else* *Factor.trad* := ... *Factor.tipo* := *simbolo*.`tipo` ... *endif*}

Tabla de símbolos (ETDS ámbitos)

ETDS para gestionar los ámbitos

S → `{tsActual = new TablaSimbolos(null)}` *SecSp*

...

Sp → *TipoFuncion* **id** **(**
`{tsActual->nuevoSimb(`**id**.*lexema*, *TipoFuncion.tipo*`)`
`tsActual = new TablaSimbolos(tsActual)}`
Args **)** *Bloque*
`{ ...`
`tsActual = tsActual->getAmbitoAnterior()`
`...`
`}`

...

Instr → `{ tsActual = new TablaSimbolos(tsActual)}`
Bloque `{ ...`
`tsActual = tsActual->getAmbitoAnterior()`
`...`
`}`

Implementación de ETDS

1. Con un analizador ascendente: hay que utilizar *marcadores* para implementar las acciones en mitad de la parte derecha y los atributos heredados (lo veremos más adelante).
2. Con un analizador descendente recursivo:
   - Los atributos sintetizados deben ser devueltos por las funciones de los no terminales (cuando hay más de un atributo es mejor devolver un `struct` o un objeto con todos los atributos de los no terminales)
   - Los atributos heredados son parámetros que se les pasan a las funciones de los no terminales
   - IMPORTANTE: puede ser necesario almacenar algunos tokens antes de llamar a la función `empareja`.
   - La traducción de la cadena de entrada es devuelta por la función que analiza el símbolo inicial de la gramática.

Implementación de ETDS (2)

Ejemplo 3 de traducción, implementado con un ASDR:

```
String D()    // D -> T id L
{
  String ttrad,idlexema,ltrad;

  ttrad = T();
  idlexema = token.lexema;
  empareja(Token.ID);
  ltrad = L(ttrad);  // L.th := T.trad

  return idlexema + ":" + ttrad + ltrad;
}
```

```
String L(String th)  // L -> coma id L  | epsilon
{
  if (token.tipo == Token.COMA)
  {
    String idlexema,ltrad;

    emparejar(Token.COMA);
    idlexema = token.lexema;
    emparejar(Token.ID);
    ltrad = L(th);  // L1.th := L.th
    return ";" + idlexema + ":" + th + ltrad;
  }
  else if (token == Token.FINFICHERO)
    return "";  // L -> epsilon { L.trad := "" }
  else
    errorSintactico(....);
}
```


Traducción de expresiones aritméticas con ETDS

- La mayoría de los operadores aritméticos son asociativos por la izquierda, por lo que se usan gramáticas recursivas por la izquierda para reflejar dicha asociatividad en el árbol sintáctico y traducir correctamente las expresiones
- Es posible realizar el mismo proceso de traducción con gramáticas LL(1), generando la traducción de la operación en el punto del árbol en que se dispone de las traducciones de los dos operandos y pasando esa traducción al resto del árbol. Por ejemplo, para traducir

  `2+3-4` `resta(suma(2,3),4)`

  cuando se está procesando el + se genera la traducción `suma(2,3)` y se pasa a otro no terminal (porque es la traducción del primer operando de la resta)

Traducción de expresiones aritméticas con ETDS (2)

`a+b-c+d`	`sum(res(sum(a,b),c),d)`
`a+b-c`	`res(sum(a,b),c)`

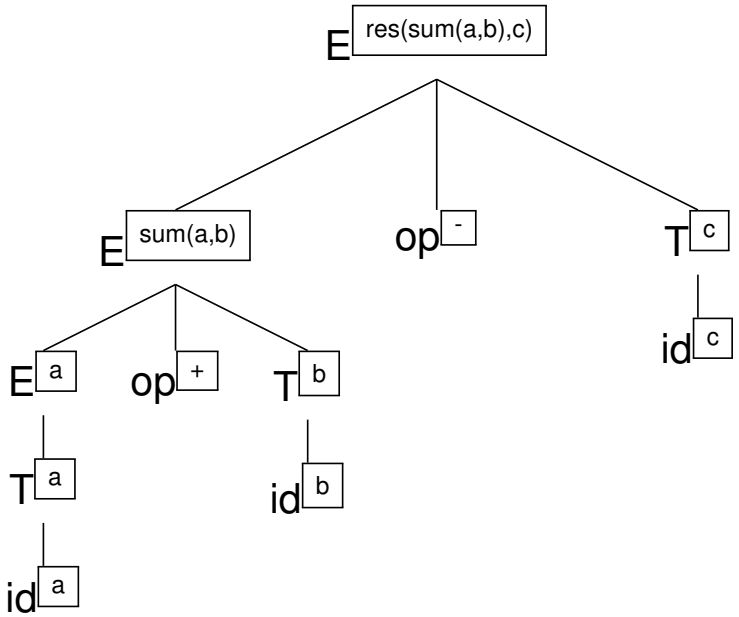
La asociatividad por la izquierda de los operadores '+' y '−' implica utilizar una gramática con recursividad por la izquierda:

$$
\begin{array}{lcl}
E & \longrightarrow & E \ \textbf{op} \ T \\
E & \longrightarrow & T \\
T & \longrightarrow & \textbf{id} \\
T & \longrightarrow & (\ E\ )
\end{array}
$$

Traducción de expresiones aritméticas con ETDS (3)

$$
\begin{array}{lcl}
E & \rightarrow & E \ \textbf{op} \ T \\
E & \rightarrow & T \\
T & \rightarrow & \textbf{id} \\
T & \rightarrow & (\ E\ )
\end{array}
$$

`a+b-c` $\Rightarrow$
`res(sum(a,b),c)`



Traducción de expresiones aritméticas con ETDS (4)

El ETDS quedaría de esta manera:

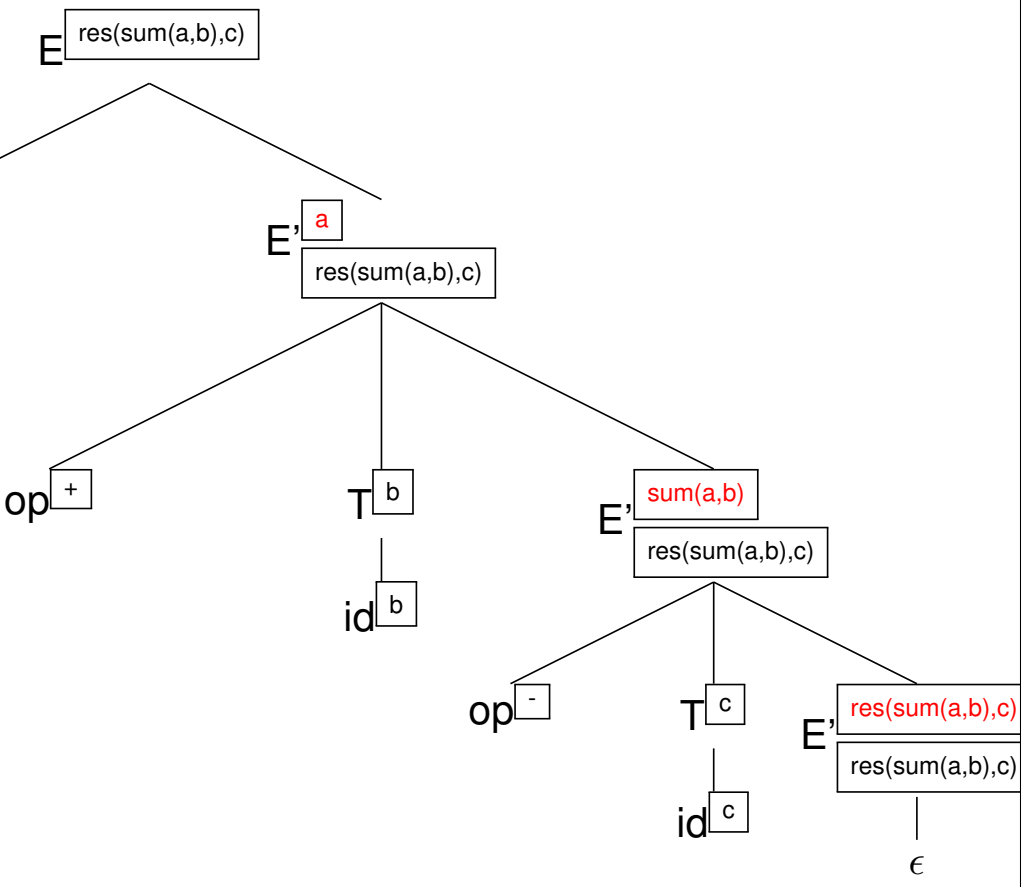
$$
\begin{array}{lcl}
E & \longrightarrow & E \ \textbf{op} \ T \ \{ E.trad := \textbf{op}.trad || \text{"("} || E_1.trad || \text{","} || T.trad || \text{")"} \} \\
E & \longrightarrow & T \ \{ E.trad := T.trad \} \\
T & \longrightarrow & \textbf{id} \ \{ T.trad := \textbf{id}.lexema \} \\
T & \longrightarrow & \textbf{(} \ E \ \textbf{)} \ \{ T.trad := E.trad \}
\end{array}
$$

Traducción de expresiones aritméticas con ETDS (5)

Si queremos realizar la misma traducción con una gramática sin recursividad por la izquierda (porque por ejemplo queremos usar un ASDR), la gramática sería:

$$
\begin{array}{lcl}
E & \longrightarrow & T \ E' \\
E' & \longrightarrow & \textbf{op} \ T \ E' \\
E' & \longrightarrow & \epsilon \\
T & \longrightarrow & \textbf{id} \\
T & \longrightarrow & \textbf{(} \ E \ \textbf{)}
\end{array}
$$

Traducción de expresiones aritméticas con ETDS (6)



$E \rightarrow T\ E'$
$E' \rightarrow \mathbf{op}\ T\ E'$
$E' \rightarrow \epsilon$
$T \rightarrow \mathbf{id}$
$T \rightarrow (\ E\ )$

`a+b-c` $\Rightarrow$
`res(sum(a,b),c)`

Traducción de expresiones aritméticas con ETDS (7)

El ETDS quedaría:

$$
\begin{array}{lcl}
E & \longrightarrow & T\ \{E'.th := T.trad\}\ E'\ \{E.trad := E'.trad\} \\
E' & \longrightarrow & \mathbf{op}\ \ T\ \{E'_1.th := \mathbf{op}.trad||"("||E'.th||","||T.trad||")"\} \\
 & & E'\ \{E'.trad := E'_1.trad\} \\
E' & \longrightarrow & \epsilon\ \{E'.trad := E'.th\} \\
T & \longrightarrow & \mathbf{id}\ \ \{T.trad := \mathbf{id}.lexema\} \\
T & \longrightarrow & (\ E\ )\ \ \{T.trad := E.trad\}
\end{array}
$$

Traducción de expresiones con enteros y reales

- La mayoría de los lenguajes permiten mezclar enteros y reales en expresiones:

```
2 + 3.5 - 7 / 2
```

- Sin embargo, la mayoría de los lenguajes objeto no permite mezclar, por lo que es necesario generar conversiones de código y operadores específicos para cada tipo (p.ej. un operador de suma para reales y otro para enteros)
- Las reglas de conversión son:
  - Si los dos operandos son del mismo tipo, no hay conversiones:

```
 2   +   3                    sumaI( 2 , 3 )
2.1 + 3.2                    sumaR( 2.1 , 3.2 )
```

  - Si un operando es real y el otro entero, se convierte el entero a real y se genera un operador real:

```
2 + 3.5                 sumaR( itor(2) , 3.5 )
2 + 3.5 - 7 / 2         restaR( sumaR( itor(2) , 3.5 ) , itor( divI( 7,2 )))
```

    Las subexpresiones enteras se convierten a real lo más tarde que sea posible

Traducción de expresiones con enteros y reales (2)

El ETDS con tipos quedaría de esta manera:

$$
\begin{array}{lcl}
E & \longrightarrow & E\ \textbf{op}\ T\ \{E.(trad, tipo) := opera(\textbf{op}, E_1.(trad, tipo), T.(trad, tipo))\} \\
E & \longrightarrow & T\ \{E.trad := T.trad\ ;\ E.tipo := T.tipo\} \\
T & \longrightarrow & \textbf{numentero}\ \{T.trad := \textbf{numentero}.lexema\ ;\ T.tipo := ENTERO\} \\
T & \longrightarrow & \textbf{numreal}\ \{T.trad := \textbf{numreal}.lexema\ ;\ T.tipo := REAL\} \\
T & \longrightarrow & \textbf{id}\ \{T.trad := \textbf{id}.lexema\ ;\ T.tipo := TDS.tipo(\textbf{id}.lexema)\}
\end{array}
$$

(en el caso de las variables, se obtiene el tipo de la tabla de símbolos)

Traducción de expresiones con enteros y reales (3)

```
func opera(op, Izq.(trad, tipo), Der.(trad, tipo))
  if(Izq.tipo == ENTERO && Der.tipo == ENTERO)
    tipo := ENTERO
    trad := op.trad||"I("||Izq.trad||","||Der.trad||")"
  elsif(Izq.tipo == REAL && Der.tipo == ENTERO)
    tipo := REAL
    trad := op.trad||"R("||Izq.trad||",itor("||Der.trad||"))"
  elsif(Izq.tipo == ENTERO && Der.tipo == REAL)
    tipo := REAL
    trad := op.trad||"R(itor("||Izq.trad||"),"||Der.trad||")"
  else // REAL && REAL
    tipo := REAL
    trad := op.trad||"R("||Izq.trad||","||Der.trad||")"
  endif

  return(trad, tipo)
```


Traducción de expresiones con enteros y reales (4)

$$E \longrightarrow T \ \{E'.(trh, tih) := T.(trad, tipo)\}\ E' \ \{E.(trad, tipo) := E'.(trad, tipo)\}$$

$$E' \longrightarrow \mathbf{op}\ T \ \{E'_1.(trh, tih) := opera(\mathbf{op}, E'.(trh, tih), T.(trad, tipo))\}\ E' \ \{E'.(trad, tipo) := E'_1.(trad, tipo)\}$$

$$E' \longrightarrow \epsilon \ \{E'.(trad, tipo) := E'.(trh, tih)\}$$

$$T \longrightarrow \ldots$$

- El no terminal E' tiene dos atributos heredados, *trh* y *tih* que tienen la traducción y el tipo del operando de la izquierda
- Antes de procesar E'_1, se opera y se obtiene la traducción y el tipo de la operación y se pasan como atributos heredados a E'_1, porque son su *operando de la izquierda*

Ejercicio 2 (sin atributos heredados)

Diseña un ETDS para traducir declaraciones con inicialización de Java a C/C++, comprobando a la vez que el tipo y número de dimensiones de declaración e inicialización coincide. Por ejemplo:

```
int[][][]   matriz = new int[10][20][30];        int matriz[10][20][30];
float[]     bidim  = new float[15][24];          error dims: 1 vs 2
int[][]     badim  = new float[1][4];            error tipos: int vs float
int[][][][] badum  = new float[4];               error tipos: int vs float
int[][]     badam  = new int[1][0];              error dim 0
```

La gramática que debes utilizar para diseñar el ETDS es la siguiente:

S	$\longrightarrow$	*Tipo DimSN* **id asig new** *Tipo Dim* **pyc**
DimSN	$\longrightarrow$	*DimSN* **cori cord**
DimSN	$\longrightarrow$	**cori cord**
Dim	$\longrightarrow$	*Dim* **cori entero cord**
Dim	$\longrightarrow$	**cori entero cord**
Tipo	$\longrightarrow$	**int**
Tipo	$\longrightarrow$	**float**
Tipo	$\longrightarrow$	**boolean**

Ejercicio 3 (examen marzo 1997) (1 de 2)

Diseña un ETDS para traducir declaraciones de funciones y procedimientos anidados en Pascal a C. El proceso de traducción se puede especificar con los siguientes ejemplos de traducción:

```
function f:integer;          int f();
   codigo
endfunc;

procedure p1;                void p1();
   procedure p2;             void p1_p2();
      codigo
   endproc;
   codigo
endproc;
```

```
procedure p1;                void p1();
  procedure p2;              void p1_p2();
    codigo                   float p1_f1();
  endproc;                   void p1_p3();
  function f1:real;          void p1_p3_p4();
    codigo
  endfunc;
  procedure p3;
    procedure p4;
      codigo
    endproc;
    codigo
  endproc;
  codigo
endproc;
```

Ejercicio 3 (examen marzo 1997) (2 de 2)

Debes diseñar el ETDS utilizando como base la siguiente gramática, que genera el lenguaje fuente:

$$
\begin{array}{lcl}
S & \longrightarrow & P \\
P & \longrightarrow & \textbf{procedure id ; } L \textbf{ endproc ;} \\
P & \longrightarrow & \textbf{function id : } T \textbf{ ; } L \textbf{ endfunc ;} \\
L & \longrightarrow & P\ L \\
L & \longrightarrow & \textbf{codigo} \\
T & \longrightarrow & \textbf{integer} \\
T & \longrightarrow & \textbf{real}
\end{array}
$$

Ejercicio 4 (1 de 2)

Dada la siguiente gramática (que permite declarar clases anidadas y sus métodos para un determinado lenguaje orientado a objetos):

$$
\begin{array}{lcl}
S & \longrightarrow & C \\
C & \longrightarrow & \textbf{class id \{ } B\ V \textbf{ \}} \\
B & \longrightarrow & \textbf{public : } P \\
B & \longrightarrow & \epsilon \\
V & \longrightarrow & \textbf{private : } P \\
V & \longrightarrow & \epsilon \\
P & \longrightarrow & D\ P \\
P & \longrightarrow & \epsilon \\
D & \longrightarrow & T \textbf{ id (} T \textbf{ id } L \textbf{)} \\
D & \longrightarrow & C \\
L & \longrightarrow & \textbf{, } T \textbf{ id } L \\
L & \longrightarrow & \epsilon \\
T & \longrightarrow & \textbf{int} \\
T & \longrightarrow & \textbf{float}
\end{array}
$$

Ejercicio 4 (2 de 2)

Construye un ETDS que traduzca a una notación como la indicada en este ejemplo:

```
class A {
  public:
    int f1(int n,float s)
  private:
    class B {
      private:
        float f2 (float r,float s,float t)
        class C {}
    }
}
```

```
clase A {
  público:
    A::f1 (entero x real -> entero)
  privado:
    clase A::B {
      privado:
        A::B::f2 (real x real x real -> real)
        clase A::B::C {}
    }
}
```


Ejercicio 5 (examen febrero 1998) (1 de 2)

Diseña un ETDS para traducir declaraciones de funciones en C a Pascal. El proceso de traducción se puede especificar con los siguientes ejemplos de traducción:

```
int f(void),
    g(float a,int *b);
```

```
function f:integer;
function g(a:real;var b:integer):integer;
```

```
void h(int a,float *c),
     j(void);
```

```
procedure h(a:integer;var c:real);
procedure j;
```

```
float f(int a);
```

```
function f(a:integer):real;
```

```
int f(int a,int b,int c),
    g(int d),
    h(int e);
```

```
function f(a:integer;b:integer;c:integer):integer;
function g(d:integer):integer;
function h(e:integer):integer;
```

Ejercicio 5 (examen febrero 1998) (2 de 2)

Debes diseñar el ETDS utilizando como base la siguiente gramática, que genera el lenguaje fuente:

$$
\begin{array}{lcl}
S & \longrightarrow & \textit{TipoFun}\ \ L\ \ \textbf{puntoycoma} \\
\textit{TipoFun} & \longrightarrow & \textbf{void} \\
\textit{TipoFun} & \longrightarrow & \textbf{int} \\
\textit{TipoFun} & \longrightarrow & \textbf{float} \\
L & \longrightarrow & F\ \ Lp \\
Lp & \longrightarrow & \textbf{coma}\ \ F\ \ Lp \\
Lp & \longrightarrow & \epsilon \\
F & \longrightarrow & \textbf{ident}\ \ \textbf{lpar}\ \ A\ \ \textbf{rpar} \\
A & \longrightarrow & \textbf{void} \\
A & \longrightarrow & \textit{Argu}\ \ M \\
M & \longrightarrow & \textbf{coma}\ \ \textit{Argu}\ \ M \\
M & \longrightarrow & \epsilon \\
\textit{Argu} & \longrightarrow & \textit{Tipo}\ \ \textbf{ident} \\
\textit{Argu} & \longrightarrow & \textit{Tipo}\ \ \textbf{asterisco}\ \ \textbf{ident} \\
\textit{Tipo} & \longrightarrow & \textbf{int} \\
\textit{Tipo} & \longrightarrow & \textbf{float}
\end{array}
$$

Ejercicio 6 (mayo 1997)

Queremos traducir declaraciones de variables, punteros y *arrays* en C a declaraciones en Pascal. Los siguientes ejemplos te pueden servir para entender el tipo de traducción a realizar:

```
int a,b7;                          var  a :integer;
                                        b7:integer;

char *c,**d;                       var  c:pointer of char;
                                        d:pointer of pointer of char;

float *f[6],**g[10][4],h;          var  f:array [0..5] of pointer of real;
                                        g:array [0..9,0..3] of pointer
                                                  of pointer of real;
                                        h:real;

int i[4][5][6];                    var  i:array [0..3,0..4,0..5] of integer;
```

Ejercicio 6 (2)

Escribid un ETDS para implementar este proceso de traducción utilizando como base la siguiente gramática (que genera solamente una declaración):

$$
\begin{array}{lcl}
D & \longrightarrow & T\ V\ L\ \mathbf{;} \\
T & \longrightarrow & \mathbf{int} \\
T & \longrightarrow & \mathbf{float} \\
T & \longrightarrow & \mathbf{char} \\
V & \longrightarrow & E \\
V & \longrightarrow & V\ \mathbf{[\ nint\ ]} \\
E & \longrightarrow & \mathbf{*}\ E \\
E & \longrightarrow & \mathbf{id} \\
L & \longrightarrow & \mathbf{,}\ V\ L \\
L & \longrightarrow & \epsilon
\end{array}
$$

Ejercicio 7 (diciembre 1996)

Queremos traducir declaraciones sencillas de variables en C a declaraciones en Pascal. Por ejemplo, la traducción de

```
int a,b7; float c; char ddd, efg
```

debería ser

```
var a,b7: integer; var c: real; var ddd, efg: char;
```

Ejercicio 7 (2)

Realiza dos ETDS para implementar este proceso de traducción utilizando como base las siguientes gramáticas:

1.

$$
\begin{array}{lcl}
S & \longrightarrow & S \ \mathbf{;} \ D \\
S & \longrightarrow & D \\
D & \longrightarrow & T \ L \\
T & \longrightarrow & \mathbf{int} \\
T & \longrightarrow & \mathbf{float} \\
T & \longrightarrow & \mathbf{char} \\
L & \longrightarrow & L \ \mathbf{,} \ \mathbf{id} \\
L & \longrightarrow & \mathbf{id}
\end{array}
$$

Ejercicio 7 (3)

2.

$$
\begin{array}{lcl}
S & \longrightarrow & D \ Sp \\
Sp & \longrightarrow & \mathbf{;} \ D \ Sp \\
Sp & \longrightarrow & \epsilon \\
D & \longrightarrow & T \ \mathbf{id} \ L \\
T & \longrightarrow & \mathbf{int} \\
T & \longrightarrow & \mathbf{float} \\
T & \longrightarrow & \mathbf{char} \\
L & \longrightarrow & \mathbf{,} \ \mathbf{id} \ L \\
L & \longrightarrow & \epsilon
\end{array}
$$

Ejercicio 8 (junio 2021)

$$
\begin{array}{lcl}
D & \longrightarrow & \textbf{var } L \\
L & \longrightarrow & L\ V \\
L & \longrightarrow & V \\
V & \longrightarrow & \textbf{id dosp } C \textbf{ pyc} \\
C & \longrightarrow & A\ C \\
C & \longrightarrow & P \\
A & \longrightarrow & \textbf{array cori } R \textbf{ cord of} \\
R & \longrightarrow & R \textbf{ coma } G \\
R & \longrightarrow & G \\
G & \longrightarrow & \textbf{numentero ptopto numentero} \\
P & \longrightarrow & \textbf{pointer of } P \\
P & \longrightarrow & T \\
T & \longrightarrow & \textbf{integer} \\
T & \longrightarrow & \textbf{real}
\end{array}
$$

Ejercicio 8 (2)

`var a :integer;`
`b7:integer;`
`b2:real;`
	`int a;`
	`int b7;`
	`float b2;`

`var c:pointer of real;` `d:pointer of pointer of integer;`	`float *c;` `int **d;`

`var f:array [0..5] of pointer of real;`
`g:array [7..9,3..3] of`
`array [1..10,2..5] of`
`of pointer`
`of pointer of integer;`
`h:array [15..18] of real;`
	`float *f[6];`
	`int **g[3][1][10][4];`
	`float h[4];`

Si en el rango el primer número es mayor que el segundo (p.ej. `7..5`), se debe dar un error semántico (que abortaría la traducción).

Ejercicio 9 (julio 2021)

$$
\begin{array}{lcl}
S & \longrightarrow & C \\
C & \longrightarrow & \textbf{class id \{ } B\ V \textbf{ \}} \\
B & \longrightarrow & \textbf{public : } P \\
B & \longrightarrow & \epsilon \\
V & \longrightarrow & \textbf{private : } P \\
V & \longrightarrow & \epsilon \\
P & \longrightarrow & D\ P \\
P & \longrightarrow & \epsilon \\
D & \longrightarrow & T \textbf{ id (} T \textbf{ id } L \textbf{)} \\
D & \longrightarrow & C \\
L & \longrightarrow & \textbf{, } T \textbf{ id } L \\
L & \longrightarrow & \epsilon \\
T & \longrightarrow & \textbf{int} \\
T & \longrightarrow & \textbf{float}
\end{array}
$$

Ejercicio 9 (2)

```
class A {
  public:
    int f1(int n,float s)
  private:
    class B {
      public:
        float f2 (float t)
      private:
        float f3 (float r,
          float s,float t)
        class C {}
    }
}
```

```
pub class A {
  pub fun A::f1 (int x real -> int)
  priv class A::B {
    pub  fun A::B::f2 (real -> real)
    priv fun A::B::f3 (real x
              real x real -> real)
    priv class A::B::C {
    }
  }
}
```