TEMA 6

ANÁLISIS SEMÁNTICO

Bibliografía:

- Aho, A.V., Sethi, R., Ullman, J.D. (1990), *Compiladores: principios, técnicas y herramientas*, Tema 5, 6 (pag. 287-400), Tema 7 (pag. 443-454).
- Louden, K.C. (1997), *Compiler Construction: Principles and Practice*, Tema 6, páginas: 257-344.

Contenido: *Duración:* 10 horas

1 La fase de análisis semántico.

2 Especificación semántica de un lenguaje:

- 2.1 Concepto de atributo, tipos de atributos y tipos de enlace.
- 2.2 Gramáticas de atributos.
- 2.3 Ecuaciones de atributos.
- 2.4 Árbol de análisis sintáctico anotado.

3 Métodos para la evaluación de atributos:

- 3.1 Basados en grafos de dependencias.
- 3.2 Basados en reglas.

4 Cálculo de atributos durante el proceso de análisis sintáctico.

5 Teorema de Knuth.

6 La Tabla de Símbolos:

6.1 Organización y mantenimiento de la Tabla de Símbolos.

6.2 Reglas de ámbito de referencia.

7 Comprobación de tipos:

7.1 Representación de expresiones de tipos mediante árboles.

7.2 Equivalencia estructural y por nombre.

8 Inferencia de tipos.

9 Errores semánticos.

10 Esquemas de traducción dirigidos por la sintaxis.

6.1 LA FASE DE ANÁLISIS SEMÁNTICO

En este tema se aborda el problema del cálculo de información que no puede ser descrita por las gramáticas independientes del contexto (GIC), y que por tanto, no se considera como parte del análisis sintáctico. La información que se calcula en esta fase está relacionada con el significado (la semántica) del programa y no con su estructura (la sintáxis).

Se asocia información a las construcciones del lenguaje de programación proporcionando atributos a los símbolos de la gramática (por ejemplo: el valor de una expresión, el tipo de una variable, su ámbito, un trozo de código, el número de argumentos de una función, etc). Los valores de los atributos se calculan mediante reglas semánticas asociadas a las producciones gramaticales.

El análisis semántico incluye:

- La construcción de la Tabla de Símbolos para llevar un seguimiento del significado de los identificadores en el programa (variables,

funciones, tipos, parámetros y método de paso de parámetros en funciones, etc)

- Realizar la comprobación e inferencia de tipos en expresiones y sentencias (por ejemplo, que ambos lados de una asignación tengan tipos adecuados, que no se declaren variables con el mismo nombre, que los parámetros de llamada a una función tengan tipos adecuados, número de parámetros correcto,)

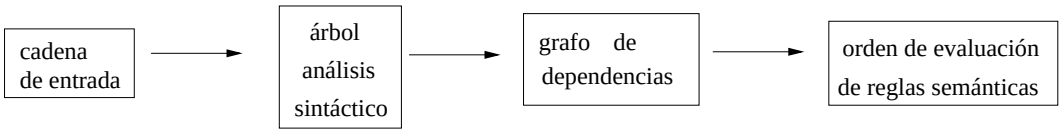
Nos centraremos en el análisis semántico estático: se realiza en tiempo de compilación, no de ejecución.

¿ Cómo vamos a especificar (describir) la estructura semántica de un lenguaje? Mediante gramáticas de atributos.

¿ Cómo vamos a implementar la estructura semántica de un lenguaje?

A partir de la construcción del árbol de análisis sintáctico, lo recorreremos en un determinado orden y calcularemos en cada nodo la información semántica necesaria (el valor de una expresión, el tipo de una variable, su ámbito de declaración, el número de argumentos de una función, etc).

Conceptualmente, se analiza sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol, en un determinado orden para tener en cuenta las dependencias, para evaluar las reglas semánticas en sus nodos.



Hay casos especiales que se pueden implantar de una sola pasada evaluando las reglas semánticas durante el proceso de análisis sintáctico, sin construir explícitamente un árbol de análisis sintáctico. Mayor eficiencia en cuánto al tiempo de compilación.

6.2 ATRIBUTOS Y GRAMÁTICAS DE ATRIBUTOS

Un **atributo** es cualquier propiedad de una construcción de un lenguaje de programación. Varían en función del tipo de información que contienen, su complejidad de cálculo y el momento en el que son calculados (en tiempo de compilación (atributos estáticos) o de ejecución (dinámicos)).

Ejemplos típicos son:

- el nombre de una variable
- el tipo de una variable
- el ámbito de una variable
- el valor de una expresión
- el número de argumentos de una función
- la posición en memoria de una variable
- un fragmento de código

Ejemplos de atributos que se calculan en tiempo de compilación: el tipo de dato, el nombre de una variable, el código de un procedimiento.

Ejemplos de atributos que se calculan en tiempo de ejecución: la posición de una variable, el valor de una expresión. Aunque algunas expresiones son constante y se pueden calcular en tiempo de compilación (cálculo previo de constantes), por ejemplo `print(''%d'', 3+4*2);`.

Nos centraremos en el análisis semántico estático (construcción de la Tabla de Símbolos y comprobaciones e inferencia de tipos), ya que el dinámico depende de la arquitectura de la máquina, del sistema operativo (control de ejecución, gestión de la memoria, llamadas a funciones, etc)

Se llama **enlace (binding)** al proceso de calcular el valor del atributo y asociarlo a su correspondiente construcción lingüística. Puede ser enlace dinámico o estático según el momento en que se calcule.

Una **gramática de atributos** es una generalización de una gramática independiente del contexto en la que cada símbolo gramatical (terminal o no-terminal) tiene asociado un conjunto de atributos.

Si X es un símbolo de la gramática, y a es un atributo asociado a X, entonces escribiremos $X.a$ para denotar el atributo a asociado a X.

Dada una producción de la forma $X_0 \rightarrow X_1 X_2 \dots X_n$, el valor de los atributos $X_i.a_j$ para cada símbolo de la gramática se relacionan entre sí mediante lo que llamaremos **ecuaciones de atributos**. Estas ecuaciones tienen la forma:

$$X_i.a_j = f_{ij}(X_0.a_1, \dots, X_0.a_k, X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

En la práctica estas funciones matemáticas f_{ij} suelen ser bastante sencillas y los atributos suelen ser independientes unos de otros (o como mucho dependientes entre varios).

Las gramáticas de atributos se suelen escribir en forma tabular, con una parte donde aparecen las producciones y otra donde aparecen las reglas semánticas (o ecuaciones de atributos) que permiten calcular los atributos.

Producción	***Reglas Semánticas***
Producción 1	Ecuaciones de atributos para producción 1
⋮	⋮
Producción n	Ecuaciones de atributos para producción n

Ejemplo 1. Consideremos la gramática para la generación de expresiones aritméticas simples: ¿ Qué atributos nos interesan en esta gramática?

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{num}$$

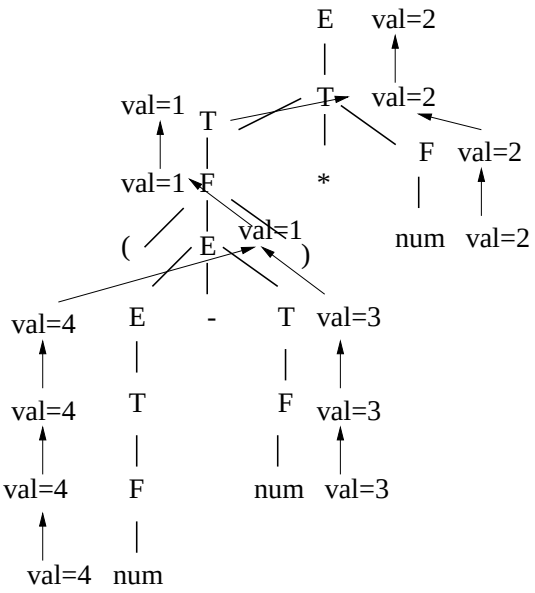
El valor numérico de la expresión. Lo denotaremos por val.

Las reglas semánticas para el cálculo de este atributo serían:

Producción	**Reglas semánticas**
$E_1 \rightarrow E_2 + T$	$E_1.val = E_2.val + T.val$
$E_1 \rightarrow E_2 - T$	$E_1.val = E_2.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T_1 \rightarrow T_2 * F$	$T_1.val = T_2.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \mathbf{num}$	$F.val = lexema(num)$

Nota: Si un símbolo aparece más de una vez en una producción, entonces cada aparición debe ser distinguida de las otras apariciones, para que los diferentes valores de las ocurrencias sean distinguidos.

Los atributos se colocan al lado de los nodos del árbol de análisis sintáctico y se dibujan las dependencias entre ellos. Una flecha que va desde los atributos que aparecen en la parte derecha de la ecuación hacia el del lado izquierdo. Por ejemplo, para la entrada `(4-3)*2`, el árbol sería:



Ejemplo 2. La gramática para la declaración de variables en C. ¿ Qué

$$Decl \rightarrow Type\ Var\_List$$
$$Type \rightarrow \mathbf{int} \mid \mathbf{float}$$
$$Var\_List \rightarrow \mathbf{id}\ ,\ Var\_List \mid \mathbf{id}$$

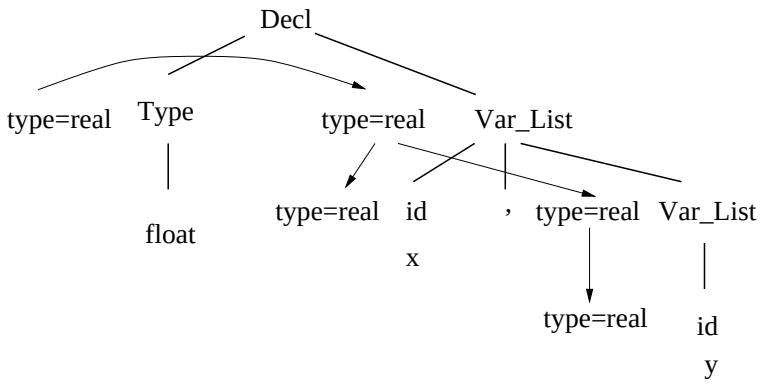
atributos nos interesan en esta gramática?

El tipo de la variable. Lo denotaremos por $type$.

Las reglas semánticas para el cálculo de este atributo serían: Para la

Producción	**Reglas semánticas**
$Decl \rightarrow Type\ Var\_List$	$Var\_List.type = Type.type$
$Type \rightarrow \mathbf{int}$	$Type.type = entero$
$Type \rightarrow \mathbf{real}$	$Type.type = real$
$Var\_List_1 \rightarrow \mathbf{id}\ ,\ Var\_List_2$	$id.type = Var\_List_1.type$ $Var\_List_2.type = Var\_list_1.type$
$Var\_List \rightarrow \mathbf{id}$	$id.type = Var\_List.type$

entrada `float x,y`, el árbol de análisis sintáctico con los atributos es:



Otro ejemplo: Como ejemplo de que un atributo puede ser cualquier cosa, por ejemplo un puntero a un nodo de árbol, veamos una gramática de atributos para la construcción de un árbol sintáctico para una gramática de expresiones aritméticas con operadores binarios:

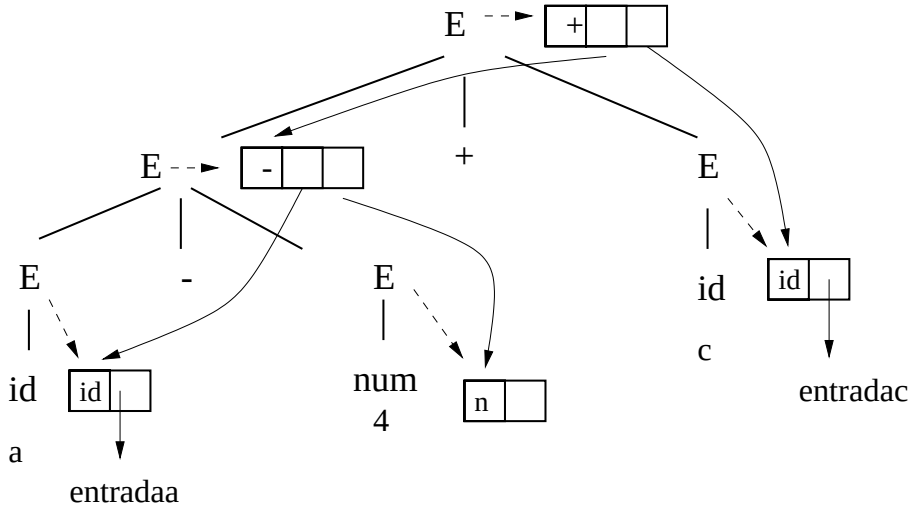
Producción	**Reglas Semánticas**
$E \to E_1 + E_2$	$E.ptr = haznodo('+', E_1.ptr, E_2.ptr)$
$E \to E_1 - E_2$	$E.ptr = haznodo('-', E_1.ptr, E_2.ptr)$
$E \to (E_1)$	$E.ptr = E_1.ptr$
$E \to \mathbf{id}$	$E.ptr = hazhoja(id.lexema, id.ptr)$
$E \to \mathbf{num}$	$E.ptr = hazhoja(num.lexema, num.val)$

- La función $haznodo(op, izda, der)$ crea un nodo para un operador con etiqueta *op* y dos campos que contienen los punteros al hijo izquierdo y derecho.
- La función $hazhoja(id, entrada)$ crea un nodo para un identificador con etiqueta **id** y un campo que contiene un puntero a la entrada de la Tabla de Símbolos para ese identificador.
- La función $hazhoja(num, val)$ crea un nodo para un número con etiqueta **num** y un campo que contiene el valor del número.

La siguiente secuencia de llamadas a funciones crea el árbol de análisis sintáctico para la expresión `a-4+c`. Los atributos $p_1, p_2, \ldots, p_5$ son punteros a los nodos y *entradaa* y *entradac* son punteros a las entradas de las variables a y c en la Tabla de Símbolos.

p_1 = hazhoja(id, entradaa)
p_2 = hazhoja(num, 4)
p_3 = haznodo('-',p_1,p_2)
p_4 = hazhoja(id, entradac)
p_5 = haznodo('+',p_3,p_4)

Son atributos sintetizados por tanto tenemos que construir el árbol de abajo hacia arriba (ascendente).



6.2.1 Tipos de atributos

Los atributos se clasifican en función de cómo se calculen en: atributos sintetizados y atributos heredados.

Atributos sintetizados: Un atributo es sintetizado si su valor depende de los valores de los atributos de sus hijos. Se podrán calcular a la vez que se realiza el análisis sintáctico ascendente LR, desde las hojas a la raíz.

Un atributo a es sintetizado si, dada una producción $A \to X_1 X_2 \dots X_n$, la única ecuación de atributos que tenemos es de la forma:

$$A.a = f(X_1.a_1, \dots, X_1.a_k, \dots, X_n.a_1, \dots, X_n.a_k)$$

Por ejemplo: el valor de una expresión en la gramática del ejemplo anterior era un atributo sintetizado.

Una gramática en la que todos los atributos son sintetizados se le llama **gramática de atributos-S**.

Atributos heredados: Un atributo es heredado si su valor depende de los valores de los atributos de su padre y/o de sus hermanos.

Por ejemplo: el tipo de una variable en la gramática anterior era un atributo heredado.

Las dependencias entre los atributos se representan en un grafo. A partir de este se obtiene un orden de evaluación de las reglas semánticas.

Árbol anotado o decorado:

Es el árbol de análisis sintáctico al que se le han añadido los atributos.

6.3 MÉTODOS PARA LA EVALUACIÓN DE LOS ATRIBUTOS

Se han propuesto varios métodos para la evaluación de las reglas semánticas:

- ***Métodos basados en grafos de dependencias***: en el momento de la compilación estos métodos obtienen un orden de evaluación a partir del grafo de dependencias sobre el árbol de análisis sintáctico para la entrada dada (el programa fuente). Poco eficientes (en espacio y tiempo) porque necesitan construir todo el árbol de análisis sintáctico y sobre él, el grafo de dependencias para cada entrada. Posibilidad de ciclos.
- ***Métodos basados en reglas***: en el momento de la construcción del compilador, para cada producción queda predeterminado por el diseñador del compilador el orden de evaluación de los atributos de esa construcción lingüística, y así la forma de recorrer el árbol para calcular ese atributo (prefija, infija, postfija). No siempre será necesario construir el árbol de análisis sintáctico para después recorrerlo, los atributos sintetizados se pueden calcular a la vez que se realiza el análisis sintáctico.

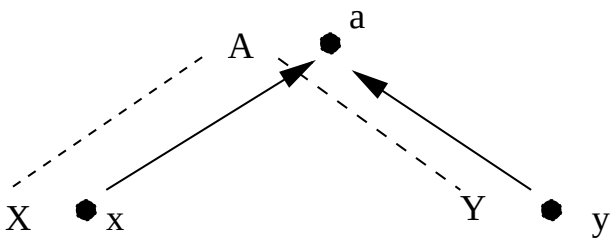
6.3.1 Grafo de dependencias

Si un atributo b en un nodo de un árbol de análisis sintáctico depende de un atributo c, entonces se debe evaluar la regla semántica para b en ese nodo después de la regla semántica que define a c. Las interdependencias entre los atributos heredados y sintetizados en los nodos de un árbol de análisis sintáctico se pueden representar mediante un *grafo de dependencias*.

¿ Cómo se construye el grafo de dependencias?

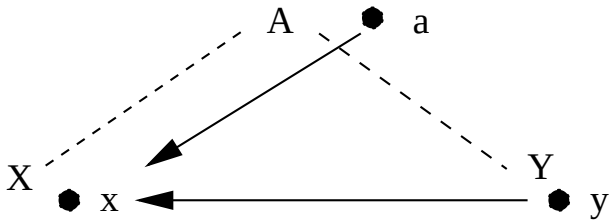
El grafo tiene un nodo por cada atributo y una arista que va desde el nodo de c hacia el nodo b, si el atributo b depende del atributo c.

Por ejemplo, si la regla $A.a = f(X.x, Y.y)$ es una regla semántica para la producción $A \to XY$. Existe un subgrafo de la forma:



Los nodos del grafo se marcan con un círculo $\bullet$ y corresponden con los atributos. Los atributos se representan junto a los nodos del árbol sintáctico.

Si la producción $A \to XY$ tiene asociada la regla semántica $X_x = g(A.a, Y.y)$, entonces habrá una arista hacia $X.x$ desde $A.a$ y también una arista hacia $X.x$ desde $Y.y$, puesto que $X.x$ depende tanto de $A.a$ como de $Y.y$.

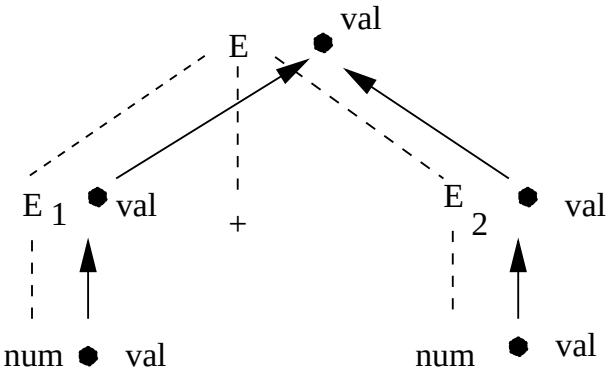


Ejemplo

Para la gramática que genera expresiones aritméticas, siempre que se use la primera producción tendremos que añadir al grafo de dependencias las aristas siguientes. Por ejemplo para la entrada:`3+2`.

Nota: en los grafos de dependencias se suelen ignorar los subíndices correspondientes a repeticiones de los símbolos de la gramática, porque

Producción	*Reglas Semánticas*
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E \rightarrow \mathbf{num}$	$E.val = lexema(num)$



queda suficientemente claro con su colocación en el árbol de análisis sintáctico a cuál símbolo nos estamos refiriendo.

Orden de evaluación de un grafo de dependencias

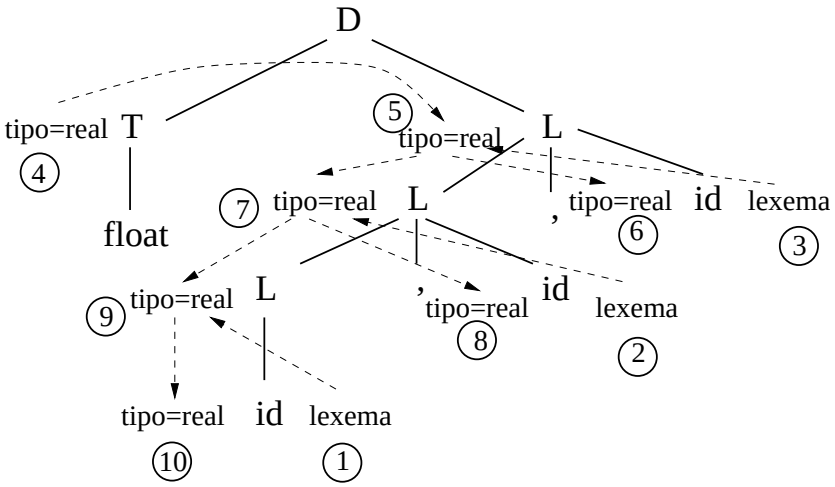
Un **ordenamiento topológico** de un grafo dirigido acíclico es todo ordenamiento $m_1, m_2, \ldots, m_k$ de los nodos del grafo, tal que garanticemos que las aristas van desde los nodos que aparecen primero en el ordenamiento a los que aparecen más tarde. Es decir, si $m_i \rightarrow m_j$ es una arista desde m_i a m_j, entonces m_i aparece antes que m_j en el ordenamiento.

Un ordenamiento topológico da un orden válido para evaluar los atributos, es decir, recorriendo el grafo en ese orden se nos garantiza que tenemos los valores de los atributos ya calculados necesarios para evaluar las reglas semánticas en cada nodo que visitemos. Es decir, los atributos $c_1, \ldots, c_k$ en una regla semántica $b = f(c_1, \ldots, c_k)$ están disponibles para poder evaluar f.

Ejemplo. Para la gramática que genera declaraciones de variables, nos interesa un atributo que indica el tipo de los identificadores.

Producción	**Reglas Semánticas**
D → T L	L.tipo = T.tipo
T → **int**	T.tipo = integer
T → **float**	T.tipo = real
L → L_1 **, id**	L_1.tipo = L.tipo añadetipo(id.lexema, L.tipo)
L → **id**	añadetipo(id.lexema, L.tipo)

El árbol de análisis sintáctico para la entrada **float id, id, id**.



El ordenamiento 1 2 3 4 5 6 7 8 9 10, sería uno de los posibles, da lugar a la ejecución de las instrucciones que almacenan el tipo *real* en la entrada de la Tabla de Símbolos para cada identificador:

a_4 = real
$a_5 = a_4$
añadetipo(lexema(id_3), a_5)
$a_7 = a_5$
añadetipo(lexema(id_2), a_7)
$a_9 = a_7$
añadetipo(lexema(id_1), a_9)

Una pregunta que surge es cómo evaluar los atributos que se encuentran en las raíces del grafo (raíz: un nodo que no tiene predecesor). El valor de los atributos en estos nodos no dependen de ningún otro atributo, y se pueden calcular usando la información disponible en ese momento (la proporciona el análisis léxico o el análisis sintáctico). Estos nodos suelen (deben) aparecer en las primeras posiciones del ordenamiento.

Desventajas: para cada entrada (cada programa fuente) tenemos que construir el grafo dirigido y encontrar en tiempo de compilación el ordenamiento topológico. Poco eficiente en espacio y tiempo. Otra alternativa a este método: la evaluación de los atributos basada en reglas.

6.3.2 Métodos basados en reglas

Es el método que se utiliza en la práctica. El implementador del compilador analiza la gramática y establece un orden de evaluación de los atributos en el momento de la construcción del compilador. Implica un recorrido prefijado del árbol de análisis sintáctico.

Recorrido del árbol con atributos sintetizados

El valor de los atributos en una gramática de atributos sintetizados puede ser calculado mediante un recorrido ascendente del árbol (*bottom-up*) o en modo postorder (*el padre el último*). En pseudocódigo:

```
procedimiento EvaluaPostOrder(T:nodo)
inicio
    para_cada hijo h de T hacer
        EvaluaPostOrder(h) // visitamos los hijos
    fin_para
        // procesamos el nod padre
    Calcular los atributos sintetizados para T
fin
```

Para lanzarlo `EvalPostOrder(raiz);`

Ejemplo. Para la gramática que genera expresiones aritméticas:

Producción	**Reglas semánticas**
$E_1 \rightarrow E_2 + E_3$	$E_1.val = E_2.val + E_3.val$
$E_1 \rightarrow E_2 - E_3$	$E_1.val = E_2.val - E_3.val$
$E_1 \rightarrow E_2 * E_3$	$E_1.val = E_2.val * E_3.val$
$E_1 \rightarrow (E_2)$	$E_1.val = E_2.val$
$E \rightarrow \mathbf{num}$	$E.val = lexema(num)$

El código correspondiente en C, con la siguiente estructura para la definición de árbol de análisis sintáctico(para esta gramática tenemos un árbol binario), sería

```
typedef enum {Plus, Minus, Mult, Num} NodeKind;
typedef struct streenode {
      NodeKind kind;
      struct streenode *lchild, *rchild;
      int val;
} STreeNode;
typedef STreeNode * SyntaxTree;
void postEval(SyntaxTree t) {
      if (t->lchild!=NULL) postEval(t->lchild);
      if (t->rchild!=NULL) postEval(t->rchild);
      switch(t->kind) {
           case Plus:
                t->val= t->lchild->val+t->rchild->val; break;
           case Minus:
                t->val= t->lchild->val-t->rchild->val; break;
           case Mult:
                t->val= t->lchild->val*t->rchild->val; break;
           case Num:
                t->val= lexema(num); break;
      }
```

```
}
```

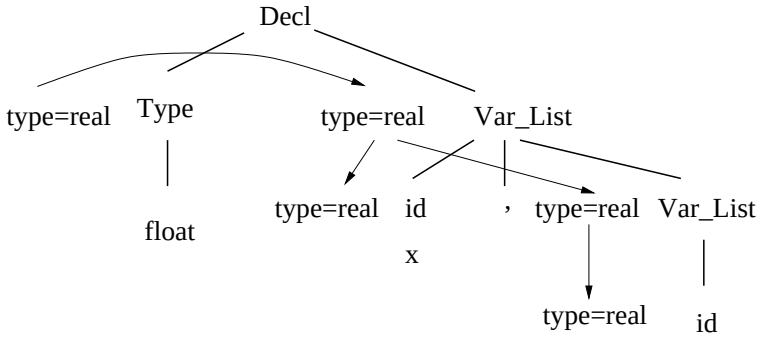
Recorrido del árbol con atributos heredados

Cuando los atributos son heredados la información pasa del padre a los hijos y/o de hermanos a hermanos. El valor de los atributos puede ser calculado mediante un recorrido del árbol en modo preorder (*procesamos el padre el primero*). A veces es necesario combinar recorrido prefijo/infijo. En pseudocódigo:

```
procedimiento EvaluaPreOrder(T:nodo)
inicio
    para_cada hijo h de T (en orden adecuado) hacer
        Evaluar los atributos heredados para h
        EvaluaPreOrder(h) //visitamos los hijos
    fin_para
fin
```

Importante: Aquí es importante el orden en que se evalúan los hijos, ya que pueden existir dependencias entre hermanos.

Ejemplo: La gramática para la declaración de variables en C.



$Decl \rightarrow Type\ Var\_List$
$Type \rightarrow \mathbf{int} \mid \mathbf{float}$
$Var\_List \rightarrow \mathbf{id}\ ,\ Var\_List \mid \mathbf{id}$

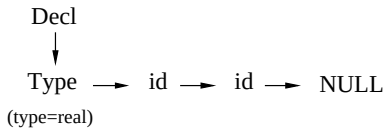
¿ Cual es el tipo de recorrido?

```
Procedure EvalType(T:nodo)
inicio
    según tiponodo de T
        Decl:
            EvalType(Type);
            Asignar tipo del hijo Type al hijo Var_List;
            EvalType(Var_List);
        Type:
            si (hijo de T == int) entonces T.type=integer;
            sino T.type=real;
        Var_List:
            Asignar T.type al primer hijo de T;
            si (tercer hijo!=NULL) entonces
                Asignar T.type al tercer hijo
                EvalType( tercer hijo Var_List);
    fin_según
fin
```

Notése como se combinan operaciones de tipo preorder e inorder dependiendo del tipo de nodo de que se trate:

- P. ej: el nodo de tipo `Decl` requiere que se calcule el tipo del primer hijo, se asigne al segundo hijo y después se evalue el segundo hijo (se realizan operaciones después de procesar el primer hijo y antes de llamar recursivamente para procesar el segundo hijo: proceso inorder).
- P. ej: el nodo de tipo `Var_List` asigna el tipo a sus hijos antes de hacer ninguna llamada recursiva (proceso preorder).

De una forma más concreta en código C, suponiendo implementado la estructura de árbol de análisis sintáctico como una lista de punteros a primer hijo y hermanos:




```
typedef enum {Decl, Type, id} NodeKind;
typedef enum {integer, real} TypeKind;
typedef struct streenode {
      NodeKind kind;
      struct streenode *child, *sibling;
      Typekind type;
      char *name; // solo para id
      } * SyntaxTree;
void EvalType(SyntaxTree t) {
      switch(t->kind) {
           case Decl:
                EvalType(t->child);
                t->child->sibling->type= t->child->type;
                EvalType(t->child->sibling);
                break;
           case Type:
                if (t->child == int) t->type=integer;
                else t->type=real;
                break;
           case id:
                if (t->sibling!=NULL) {
                     t->sibling->type= t->type;
                     EvalType(t->sibling);
                } break;
      }
}
```

Recorrido del árbol con atributos heredados y sintetizados

En gramáticas que combinan atributos sintetizados y heredados, en el caso de que los atributos heredados no dependan de atributos sintetizados es posible calcularlos en un sólo recorrido combinando `PostEval` y `PreEval`. En pseudocódigo:

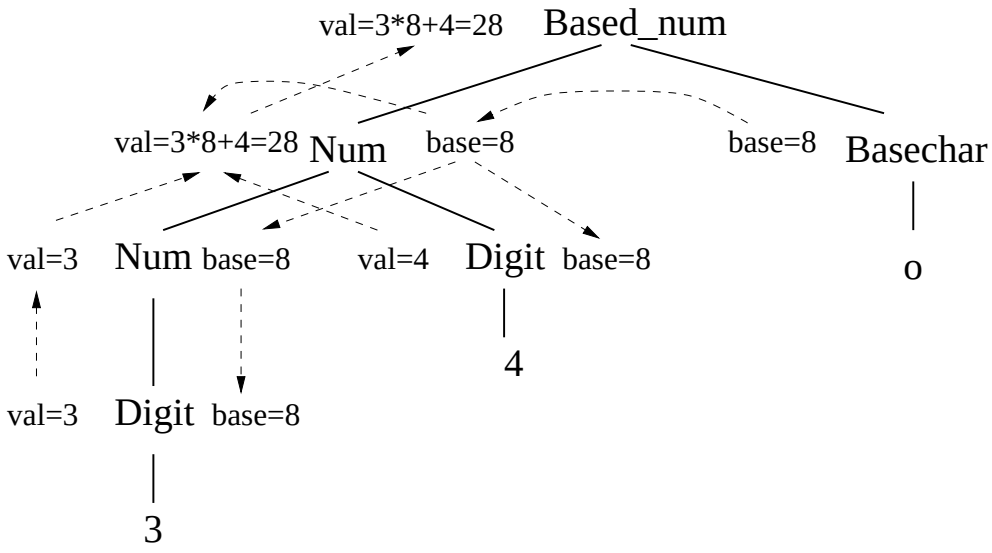
```
procedimiento EvaluaCombinada(T:nodo)
inicio
    para_cada hijo h de T hacer
        Evaluar los atributos heredados para h
        EvaluaCombinada(h)
    fin_para
    Calcular los atributos sintetizados para T
fin
```

La principal consideración a tener en cuenta durante el recorrido del árbol es que los atributos heredados en un nodo se calculen antes de que el nodo sea visitado por primera vez y que los atributos sintetizados se calculen antes de abandonar el nodo.

Ejemplo. Gramática que genera números en notación octal y decimal.

Producción	**Reglas semánticas**
Based_num → Num Basechar	Based_num.val=Num.val Num.base=Basechar.base
Basechar → **o**	Basechar.base=8
Basechar → **d**	Basechar.base=10
Num_1 → Num_2 Digit	Num_1.val= **if** (Digit.val==error or Num_2.val==error) error **else** Num_2.val*Num_1.base+Digit.val Num_2.base=Num_1.base Digit.base=Num_1.base
Num → Digit	Num.val=Digit.val Digit.base=Num.base
Digit → **0**	Digit.val=0
⋮	⋮
Digit → **8**	Digit.val= **if** (Digit.base==8) error **else** 8
Digit → **9**	Digit.val= **if** (Digit.base==8) error **else** 9

Necesitamos dos atributos: val (sintetizado) para almacenar el valor del número y $base$ (heredado) para almacenar el tipo de base en que se expresa y no depende del anterior. Para la entrada `34o`, el árbol anotado es:



El procedimiento a diseñar para la evaluación sería en este caso:

```
Procedure EvalBasedNum(t:nodo) {
   switch(t->kind) {
     Based_num:
       EvalBasedNum(t->rchild); //evalua hijo drcho
       t->lchild->base=t->rchild->base; // es hered.
       EvalBasedNum(t->lchild);//evalua hijo izqdo
       t->val=t->lchild->val; // es sintet.
     Num:
       t->lchild->base=t->base;
       EvalBasedNum(t->lchild); //evalua hijo izqd
       if (t->rchild!=NULL)
         t->rchild->base=t->base;
         EvalBasedNum(t->rchild);
         if (t->lchild->val!=error and t->rchild->val!=error
           t->val=t->base*t->lchild->val+t->rchild->val;
         else t->val=error;
       else t->val=t->lchild->val;
     Basechar:
       if (t->lchild==o) t->base=8;
       else t->base=10;
     Digit:
       if (t->base==8 and t->lchild==8 or 9) t->val=error;
       else t->val=t->lchild->val;
   }
}
```

Se ha supuesto que cuando sólo hay un hijo, éste es el izquierdo.

Situaciones en los que los atributos heredados dependen de los sintetizados son más complejas y requieren más de una pasada sobre el árbol de análisis sintáctico. Veremos un ejemplo en los ejercicios.

Aspectos a tener en cuenta en la colocación de las reglas para la evaluación de los atributos

- Un atributo heredado para un símbolo en el lado derecho de una producción se debe calcular en una acción situada antes que dicho símbolo.
- Una acción no debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.
- Un atributo sintetizado para el no-terminal de la izquierda sólo se puede calcular después de que se hayan calculado todos los atributos a los que se hace referencia. La acción para calcularlo se debe colocar al final del lado derecho de la producción.

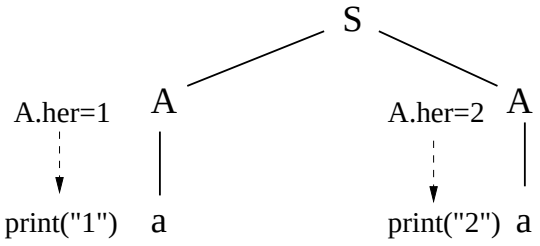
Por ejemplo, esto sería incorrecto:

$$\begin{array}{ll} S \rightarrow\ A_1\ A_2 & \{A_1.her = 1; A_2.her = 2;\} \\ A \rightarrow\ a & \{print(A.her);\} \end{array}$$

Resulta que el atributo $A.her$ es heredado y si hacemos un recorrido en modo preorden, en la segunda producción no está definido cuando se intenta imprimir su valor. La solución correcta sería:

$$\begin{array}{l} S \rightarrow\ \{A_1.her = 1;\}\ A_1\ \{A_2.her = 2;\}\ A_2 \\ A \rightarrow\ a\ \{print(A.her);\} \end{array}$$

Para la cadena aa



6.4 EVALUACIÓN DE ATRIBUTOS DURANTE EL ANÁLISIS SINTÁCTICO

¿ Hasta qué punto los atributos se pueden calcular en el propio proceso de análisis sintáctico sin necesidad de hacer recorridos adicionales sobre el árbol? La respuesta depende del tipo de análisis que se utilice.

Interés: sobre todo en los primeros compiladores con limitados recursos de computación se pretendía realizar el número menor posible de pasadas. Actualmente, no es importante.

Los métodos LL y LR, al evaluar la cadena de componentes léxicos de izquierda a derecha, implican que no existan dependencias hacia atrás en el árbol (dependencias que apuntan desde la derecha hacia la izquierda). Las gramáticas de atributos que satisfacen esta condición se les llama **gramáticas de atributos-L** (gramática de atributos por la izquierda).

Una gramática con atributos $a_1, \ldots, a_k$ es de **atributos-L** si, para cada atributo heredado a_j y cada producción $X_0 \to X_1 X_2 \ldots X_n$, las ecuaciones de atributos asociadas son de la forma:

$$X_i.a_j = f_{ij}(X_0.a_1, \ldots, X_0.a_k, \ldots, X_{i-1}.a_1, \ldots, X_{i-1}.a_k)$$

Es decir, el valor de a_j en X_i depende sólo de los atributos de los (i-ésimos) símbolos $X_0, \ldots, X_{i-1}$, que aparecen a la izquierda de X_i en la producción. De la definición se deduce que una gramática de atributos-S es de atributos-L.

Ejemplos: todas las gramáticas vistas hasta ahora, excepto la gramática que generaba números en base octal/decimal, son de atributos-L.

Dada una gramática de atributos-L, tal que los atributos heredados no dependen den los sintetizados, se pueden calcular durante el proceso de análisis sintáctico, considerando los atributos heredados como parámetros de los procedimientos y los atributos sintetizados como valores devueltos.

De este modo, se evita almacenarlos en cada nodo. La función para un no-terminal A toma como argumentos un nodo y los valores de los atributos heredados para A y devuelve como resultados los valores de los atributos sintetizados para A (de hecho, esto es lo que permite la herramienta PCCTS).

Ejemplo: La gramática que genera números en base octal y decimal. El código visto se puede convertir usando funciones en:

```
int EvalBasedNum(t:nodo) {
     return (EvalNum(t->lchild, EvalBase(t->rchild);
}

int EvalBase(t:nodo){
     if (t->lchild==o) return (8);
     else return (10);
}

int EvalNum(nodo t, int base) {
int temp,temp2;
     switch(t->kind) {
        Num:
           temp=EvalNum(t->lchild,base);
           if (t->rchild!=NULL)
              temp2=EvalNum(t->rchild, base);
              if (temp!=error and temp2!= error)
                 return (temp*base+temp2);
              else return (error);
           return(temp);
        Digit:
           if (base==8 and t->lchild->val==8 or 9)
              return (error)
           else return (t->lchild->val);
     }
```

```
}
```

Si los atributos son sintetizados se pueden evaluar con un analizador sintáctico ascendente. El analizador sintáctico puede mantener en la pila de análisis sintáctico los valores de los atributos asociados con los símbolos de la gramática. Cuando se hace una reducción se calcula el valor de los nuevos atributos a partir de los valores de los atributos de los símbolos de la parte derecha, que ya se encuentran en la pila.

Ejemplo

Dispondremos, además de la pila de análisis sintáctico, de una pila de valores en la que se van almacenando los valores de los atributos sintetizados (como un registro si hay más de uno). La pila de valores se manipula de forma paralela a la de análisis sintáctico.

$$E \rightarrow E + E \mid E * E \mid \mathbf{n}$$

PILA	ENTRADA	ACCIÓN SINT.	PILA VAL.	ACCIÓN SEMÁNT.
#	3*4+5#	desplazar	#	
#n	*4+5#	reducir $E \rightarrow n$	#3	$E.val = n.val$
#E	*4+5#	desplazar	# 3	
#E*	4+5#	desplazar	# 3	
#E*n	+5#	reducir $E \rightarrow n$	#3 4	$E.val = n.val$
#E*E	+5#	reducir $E \rightarrow E * E$	#12	$E_1.val = E_2.val * E_3.val$
#E	+5#	desplazar	#12	
#E+	5#	desplazar	#12	
#E+n	#	reducir $E \rightarrow n$	# 12 5	$E.val = n.val$
#E+E	#	reducir $E \rightarrow E + E$	#17	$E_1.val = E_2.val + E_3.val$
#E	#	aceptar	#17	

En *Bison/Yacc*, la regla semántica se hubiera implementado como:

```
E : E + E {$$=$1+$3;}
```

6.4.1 El uso de estructuras externas para almacenar los atributos

Existen algunos atributos de especial importancia cuyos valores se utilizan en diferentes puntos durante la compilación. Por ejemplo: el tipo de una variable (en la comprobación de tipos, en la generación de código). Es conveniente tenerlos almacenados de forma fácilmente accesible (en variables globales, tablas, ...) y no como información en el nodo.

La Tabla de Símbolos es el ejemplo más importante de una estructura de datos adicional al árbol de análisis sintáctico, que almacena atributos asociados a la declaración de constantes, variables, funciones y tipos.

Ejemplo de uso de estas estructuras externas: la gramática de la declaración de variables en C. Supongamos que existe ya una tabla creada en la que insertamos el nombre del identificador (que se utiliza como clave) y su tipo. En vez de almacenar el tipo de cada variable en el árbol de análisis sintáctico, lo insertamos en la Tabla de Símbolos. Además, como cada declaración tiene asociada un sólo tipo, podemos usar una variable global, $type$.

Producción	**Reglas semánticas**
$Decl \rightarrow Type\ Var\_List$	
$Type \rightarrow$ **int**	$type = entero$
$Type \rightarrow$ **float**	$type = real$
$Var\_List \rightarrow$ **id** , Var_List	insert(lexema(id),type)
$Var\_List \rightarrow$ **id**	insert(lexema(name),type)

Nota: Esto ya no es una gramática de atributos!!! (aunque el resultado sea el mismo). Ganamos en sencillez.

6.5 TEOREMA DE KNUTH (1968)

El tipo de atributo (y por tanto su método de evaluación) depende de la estructura de la gramática. Así, modificaciones a la gramática (siempre generando el mismo lenguaje) pueden dar lugar a que el cálculo de los atributos sea más sencillo.

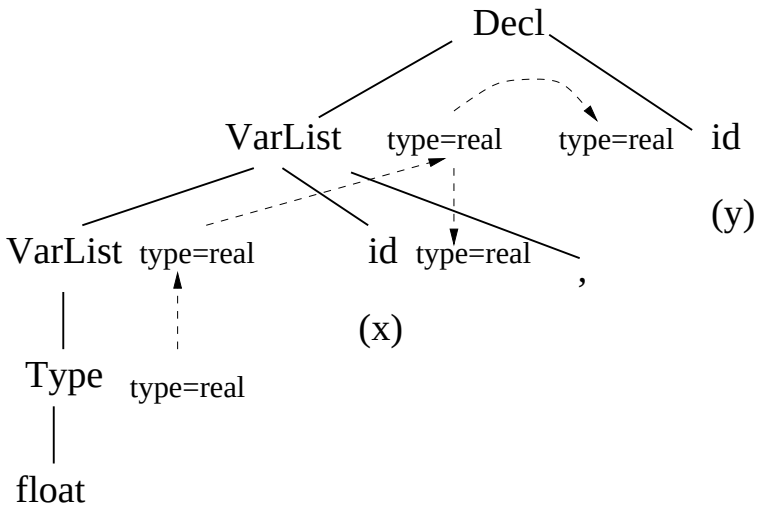
> Teorema: *Dada una gramática de atributos, todos los atributos heredados pueden convertirse en atributos sintetizados por transformación de la gramática sin modificar el lenguaje generado.*

Por ejemplo: Dada la gramática para generar declaraciones en C (recordemos que el tipo era un atributo heredado):

Producción	**Reglas semánticas**
$Decl \rightarrow Type\ Var\_List$	$Var\_List.type = Type.type$
$Type \rightarrow$ **int**	$Type.type = entero$
$Type \rightarrow$ **float**	$Type.type = real$
$Var\_List_1 \rightarrow$ **id** , Var_List_2	$id.type = Var\_List_1.type$ $Var\_List_2.type = Var\_list_1.type$
$Var\_List \rightarrow$ **id**	$id.type = Var\_List.type$

Se puede reescribir de la forma:

Producción	**Reglas semánticas**
$Decl \rightarrow Var\_List$ **id**	$id.type = Var\_List.type$
$Var\_List_1 \rightarrow Var\_List_2$ **id ,**	$id.type = Var\_List_2.type$ $Var\_List_1.type = Var\_List_2.type$
$Var\_List \rightarrow Type$	$Var\_List.type = Type.type$
$Type \rightarrow$ **int**	$Type.type = entero$
$Type \rightarrow$ **float**	$Type.type = real$



Para la entrada `float x, y`.

A primera vista se podría suponer que el atributo $type$ sigue siendo heredado. Sin embargo, aquí siempre el valor se transmite una vez (no recursivo) de un padre a una hoja (que no lo transmite a nadie) y por tanto no es visto como atributo heredado.

Este teorema es menos útil de lo que parece. Los cambios en la gramática, para convertir atributos heredados en sintetizados, a menudo hacen que las reglas sean más complejas y difíciles de entender. Los problemas con la evaluación de los atributos suelen ser un síntoma de un mal diseño de la gramática, la mejor solución es rehacer su diseño modificando la sintaxis.

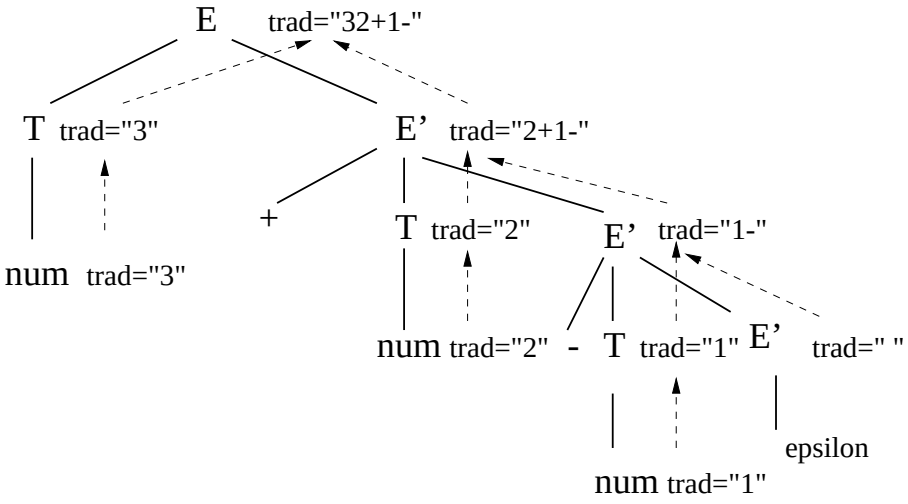
6.6 ESQUEMAS DE TRADUCCIÓN DIRIGIDOS POR LA SINTAXIS (ETDS)

Un *esquema de traducción dirigido por la sintaxis (ETDS)* permite traducir un lenguaje en otro. Se trata de una gramática independiente del contexto (que representa el lenguaje fuente) con un atributo que representa el código traducido y las reglas semánticas correspondientes para realizar la traducción. La evaluación del atributo (suele ser un atributo sintetizado) se realiza durante el propio proceso de análisis sintáctico, de ahí que se les llame *dirigidos por la sintaxis*.

Ejemplo: un ETDS que traduce expresiones en notación infija a postfija. (El símbolo || indica concatenación de cadenas). El atributo lo llamaremos $trad$, y almacena la expresión traducida.

E $\rightarrow$ T E' { E.trad = T.trad || E'.trad}
E' $\rightarrow$ + T E'$_1$ { E'.trad = T.trad || "+" || E'$_1$.trad}
E' $\rightarrow$ - T E'$_1$ { E'.trad = T.trad || "-" || E'$_1$.trad}
E' $\rightarrow \epsilon$ { E'.trad=" " }
T $\rightarrow$ **num** { T.trad=num.val }

Para la entrada `3+2-1`.



$trad$ es un atributo sintetizado (las acciones se ejecutan al final de la pro-

ducción), recorrido postfijo.

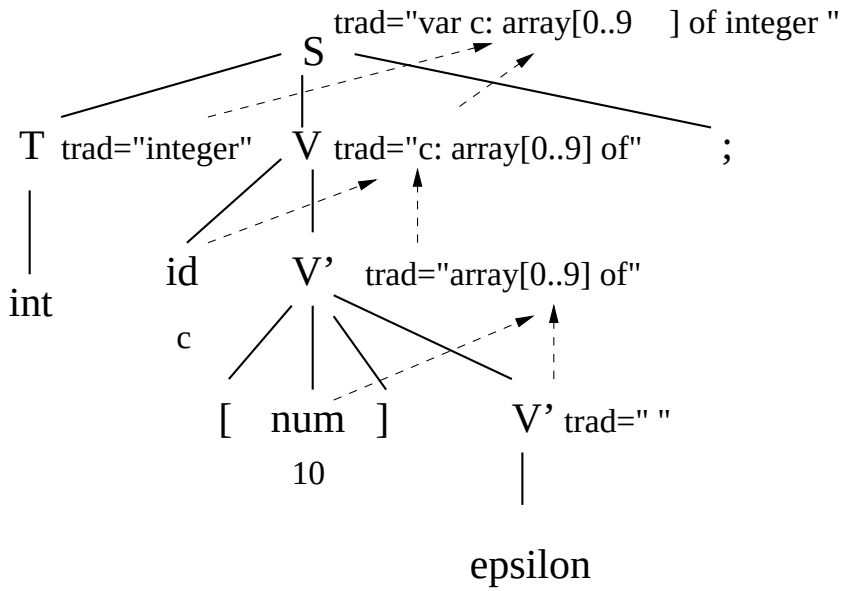
Ejemplo: traducción de declaración de variables de C a Pascal. Por ejemplo las cadenas:

int c[10];	var c: array[0..9] of integer ;
float d[5][6];	var d: array[0..4] of array [0..5] of real;
char e;	var e: char;

Gramática básica:

S $\rightarrow$ T V ;	S.trad='var' || V.trad || T.trad || ';'
V $\rightarrow$ **id** V'	V.trad = id.lexema || ':' || V'.trad
V' $\rightarrow$ [num] V'$_1$	V'.trad= 'array [0..' || num.val-1 || '] of' || V'$_1$.trad
V' $\rightarrow \epsilon$	V.trad=' '
T $\rightarrow$ **int**	T.trad='integer'
T $\rightarrow$ **float**	T.trad='float'
T $\rightarrow$ **char**	T.trad='char'

Para la entrada `int c[10];`.



6.7 LA TABLA DE SÍMBOLOS

La Tabla de Símbolos es después del árbol de análisis sintáctico la principal estructura de datos de un compilador. La Tabla de Símbolos interacciona con el analizador léxico y con el analizador sintáctico, a pesar que hemos retrasado su introducción hasta ahora. Ambos introducen información o necesitan consultarla durante todo el proceso de la compilación. También interacciona con el análisis semántico, como en la comprobación de tipo y con el módulo de generación de código (dirección de memoria, tipo de variable y tipo de operación a realizar...).

La información que se introduce en la Tabla de Símbolos depende del lenguaje. Incluye información sobre los tipos, ámbitos (*scope*) y posición de memoria. Algunos lenguajes mantienen una única Tabla de Símbolos para todas las declaraciones: de constantes, variables, tipos y funciones. Otros mantienen tablas diferentes.

El compilador utiliza la Tabla de Símbolos para llevar un seguimiento de los identificadores (variables, funciones, tipos) que aparecen en el programa fuente. Se examina la Tabla de Símbolos cada vez que se encuentra un identificador. Operaciones de consulta, inserción y borrado con bastante frecuencia.

- *inserción:* introducción de la información de declaraciones de constantes, variables, funciones y tipos.
- *búsqueda:* para obtener información de un identificador (su nombre, tipo, ámbito, ..) cuando aparece a lo largo del programa.
- *borrado:* eliminación de declaraciones (de constantes, variables, funciones o tipos) cuando nos salimos del ámbito en que se aplican.

6.7.1 Estructura de la Tabla de Símbolos

La estructura de datos elegida debe ser eficiente ante estas tres operaciones. En la práctica se utilizan:

- Listas lineales: fácil de implementar, pobre rendimiento en consultas cuando el número de entradas es grande.
- Tablas de dispersión: difícil de implementar, más espacio, pero mayor rendimiento. Es la que se utiliza en la práctica.

Otras estructuras posibles son los árboles binarios de búsqueda, árboles AVL, árboles B. Aunque no se suelen utilizar por la complejidad de la operación de borrado. Es útil que se pueda aumentar dinámicamente el tamaño de la Tabla de Símbolos durante la compilación. De lo contrario, debería ser muy grande para albergar cualquier programa fuente.

Cada entrada de la Tabla de Símbolos corresponde a la declaración de un nombre. La entrada se crea cuando aparece por primera vez el nombre, aunque la información asociada, como el tipo de identificador (variable, función, expresión de tipo), ámbito, tipo de dato, número de parámetros en funciones, etc, se rellena conforme se va conociendo a lo largo del proceso de análisis sintáctico, o incluso más tarde como la posición de memoria, que se ligará a los nombre durante la ejecución (variables estáticas o dinámicas).

¿ Qué información se introduce?

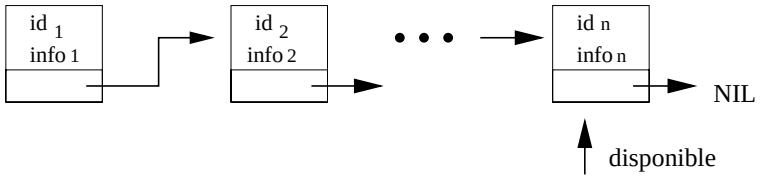
- En las *constantes*: el valor.
- En los *tipos*: la estructura del tipo, campos, tamaños, etc
- En las *variables*: su tipo, ámbito (región del programa en la que tiene validez), posición de memoria.
- Procedimientos y funciones: tipo y número de argumentos, valor de retorno, ámbito de aplicación, etc.

La Tabla de Símbolos como una lista enlazada

Los identificadores se añaden a la lista en el orden que aparecen. La búsqueda se realiza hacia atrás desde el final hasta el comienzo. Si se llega al comienzo y no se ha encontrado: fallo.

Si la tabla contiene n entradas (símbolos). Coste inserción: O(n) (tenemos que buscar primero para ver que ese nombre no existe ya). Coste consulta: O(n)

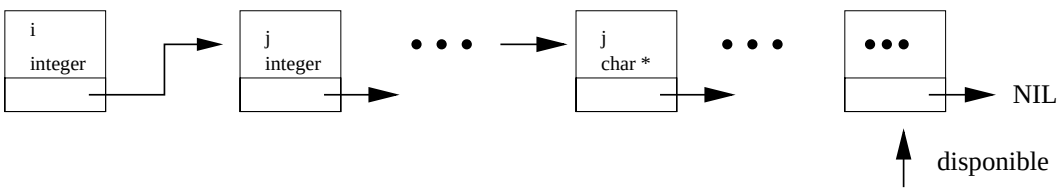
Por tanto, el coste para realizar n inserciones y realizar e consultas es a lo sumo $n(n+e)$. Si el programa es de tamaño medio, se puede tener $n = 100$ y $e = 1000$, tenemos 1000000 operaciones. Si aumenta n y e en un factor de 10, el coste se hace ya prohibitivo.



¿Cómo se implanta la regla de ámbito?

En un lenguaje estructurado por bloques, cuando se busca un identificador en la tabla debe devolverse la entrada adecuada.

```
int i,j;
...
     { char * j;
          ...
     }
```



Normalmente las entradas se insertan en orden en que aparecen. Así, si buscamos desde atrás, cuando aparezca un identificador en el programa y lo busquemos en la Tabla de Símbolos nos encontraremos primero con el que ha sido más cercano, el adecuado según la regla del anidamiento más cercano.

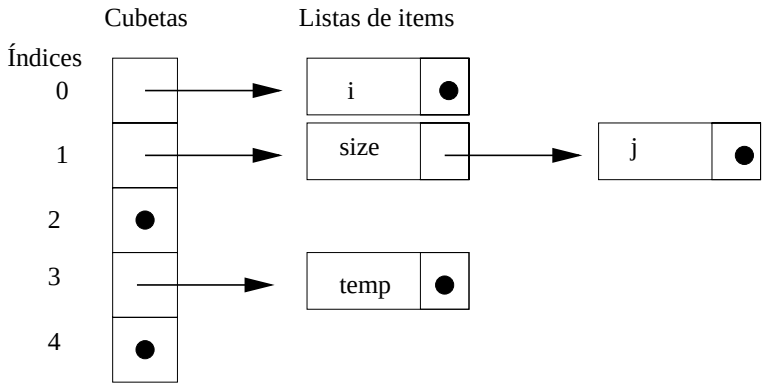
La Tabla de Símbolos como una tabla de dispersión

Se considera la *dispersión abierta*, no hay límite en el número de entradas que puede contener la tabla. Simplemente cuando se produce una colisión (dos identificadores con el mismo valor de la función de dispersión) se añade a la lista correspondiente. Permite la realización de e consultas sobre n nombres en un tiempo proporcional a $\frac{n(n+e)}{m}$, donde m es una constante elegida. El espacio ocupado aumenta con m. Compromiso entre tiempo de consulta y espacio.

Una tabla de dispersión (*Hash*) está formada por:

- Una vector de punteros de tamaño m indexado por un entero (cubetas).
- m listas enlazadas independientes unas de otras (algunas de las cuales pueden estar vacías).

Por ejemplo una tabla de tamaño 5:



Para buscar un identificador de nombre la cadena s (clave de la tabla), se le aplica la *función de dispersión h(s)*, que devuelve un entero entre 0 y $m-1$, y que es el índice del puntero que apunta al inicio de la lista en la que se debería encontrar ese identificador.

La lista promedio consta de $\frac{n}{m}$ registros suponiendo que hay n identificadores en una tabla de tamaño m. Si se elige m de modo que $\frac{n}{m}$ esté limitado por una constante, por ejemplo 2, se puede demostrar que el tiempo para acceder a una entrada es prácticamente constante. Para programas de tamaño medio, un valor de m de varios cientos es suficiente.

¿ Cómo diseñar una función de dispersión fácil de evaluar y que distribuya los identificadores uniformemente entre las listas?

1 Cada carácter $c_1, c_2, \ldots, c_k$ se convierte en un entero positivo. Por ejemplo: su código ASCII.

2 Estos valores se combinan en un único número entero, h. Una forma posible es sumar los valores de los caracteres.

3 A partir de ese entero, obtener un número comprendido entre 0 y $m-1$. Por ejemplo, dividiendo h/m y tomando el resto. Se ha comprobado que si m es un número primo, la función resto (**mod**) proporciona valores uniformemente distribuidos.

Consideraciones: es conveniente tomar todos los caracteres al calcular h. Situaciones conflictivas:

- si cogemos sólo algunos primeros o alguno últimos, problemas porque los programadores tienden a usar nombres como `temp1,temp2,m1tmp,m2tm` Demasiadas colisiones.
- si cogemos todos y los sumamos directamente, problemas. Por ejemplo, `tempx,xtemp` o cualquier permutación.

Solución: coger todos los caracteres y multiplicarlos por un peso función de su posición en la cadena.

$$h = (\alpha^{k-1}c_1 + \alpha^{k-2}c_2 + \ldots + \alpha c_{k-1} + c_k) \ \mathbf{mod} \ \ m$$

$$h = (\sum_{i=1}^{k} \alpha^{k-i}c_i) \ \mathbf{mod} \ \ m$$

donde α potencia de 2 para que las multiplicaciones se implementen de forma eficiente (16, 128...). Valores tipo de m varían entre varios cientos a mil. Se puede incrementar dinámicamente en tiempo de compilación para programas muy extensos.

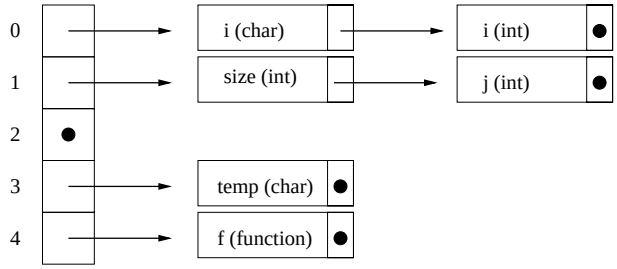
Mantenimiento del ámbito

Las reglas de ámbito varían de un lenguaje a otro, pero hay varias que son comunes: declaración antes de uso y el anidamiento más cercano para estructuras de bloque.

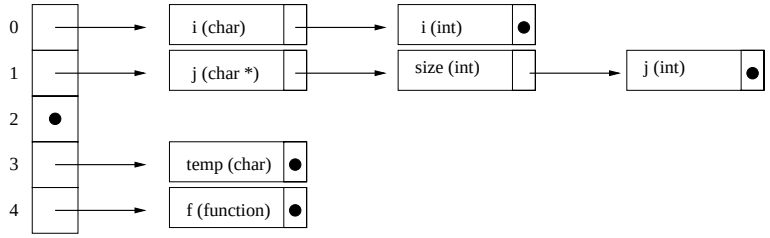
- **Declaración antes de uso**: declaración de los identificadores antes de hacer referencias a él en el texto del programa. Esta condición permite que la Tabla de Símbolos se pueda construir durante el proceso de análisis sintáctico y hacer las búsquedas tan pronto como aparezca el nombre y emitir el correspondiente mensaje de error en caso de fallo. Hace factible los compiladores en única pasada. En lenguajes como Modula-2 es necesario una primera pasada independiente para construir la Tabla de Símbolos.

- **Estructura de bloque**: un bloque es visto como cualquier construcción que contiene declaraciones. Por eje.: procedimientos, funciones, sentencias compuestas entre llaves, estructuras, uniones y las clases en C++. Dadas dos declaraciones con el mismo nombre en bloques anidados, la declaración que se aplica es la del bloque anidado más cercano.

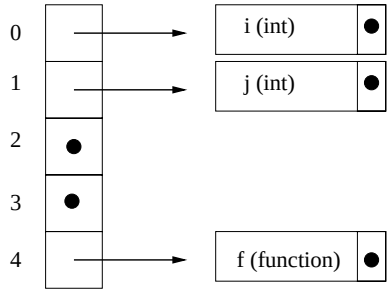
```
int i,j;
int f(int size)
{ char i, temp;
      ...
      { double j;
           ...
      }
      ...
      { char * j;
           ...
      }
}
```



Después de procesar la declaración del cuerpo de f



Después de procesar la declaración de la segunda sentencia anidada dentro de f



Después de salir de f

Para implementar la posibilidad de anidamientos, la operación *insertar* no debe sobreescribir las declaraciones previas, sino que sólo debe ocultarlas temporalmente, para que la operación de búsqueda, sólo encuentre la declaración más reciente. La operación borrado, debe también sólo borrar la más reciente (el proceso de inserción y borrado del ámbito se puede asimilar al comportamiento de una pila).

Programa → Decls ; Insts
Decls → DeclVars DeclFuncs
DeclVars → Variable DeclVars |ϵ
Variable → **var** Type **id ;**
DeclFuncs → Funcion DeclFuncs |ϵ
Funcion → **function id (** Arg LArg **):** Type **;** Decls **begin** Insts **end**
Type → **int** | **float** | **void**
Arg → Type **id** | Type *** id**
LArg → **,** Arg LArg | ϵ
Insts → **id =** E **;** Insts | ϵ
E → **id** | **num** | E + E

Para la entrada:

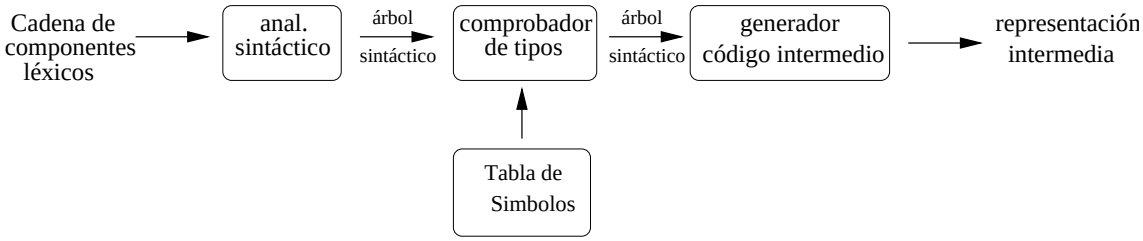
```
var int a;
var int b;
function swap(int a, int b):void;
var int temp;
begin
      temp=a;
      a=b;
      b=temp;
end
```

A cada estructura de bloque se le asigna un nivel de anidamiento (profundidad). Insertar reglas semánticas y árbol.

6.8 COMPROBACIÓN DE TIPOS

Un *comprobador de tipos* tiene como función asegurar que el tipo de una construcción coincida con el previsto en el contexto. Por eje.: llamada a una función con argumentos adecuados, indexación de sólo variables definidas como matrices,... La comprobación de tipos se puede realizar junto con el proceso de análisis sintáctico en la mayoría de lenguajes. En otros es necesario una pasada adicional (Ada) independiente.

El diseño de un comprobador de tipos se basa en información acerca de las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tipos a las construcciones del lenguaje.



6.8.1 Expresiones de tipos y su representación

Formalmente un *tipo de dato* es un conjunto de valores con ciertas operaciones sobre ellos. El tipo de una construcción del lenguaje lo denotaremos mediante una *expresión de tipo*: un tipo básico o el resultado de la aplicación de un constructor de tipos a otras expresiones de tipo.

Clases de expresiones de tipo:

1 Un tipo básico: *boolean, char, integer, real, error_tipo*.

2 El nombre de un tipo es una expresión de tipo.

3 Un constructor de tipo aplicado a expresiones de tipo es una expresión de tipo. Constructores:

- matrices: Si T es una expresión de tipo, entonces $array(I, T)$ es una expresión de tipo que indica el tipo matriz con elementos de tipo T y el conjunto de índices I.
- productos: Si T_1 y T_2 son expresiones de tipo, entonces su producto cartesiano $T_1 \times T_2$ es una expresión de tipo.
- registros: record ($T_1 \times \ldots \times T_n$)
- punteros: Si T es una expresión de tipo, entonces $pointer(T)$ es una expresión de tipo que indica el tipo "apuntador a un objeto de tipo T".
- funciones: transforma elementos de un conjunto (el dominio) a otro conjunto (el rango). $D \rightarrow R$.

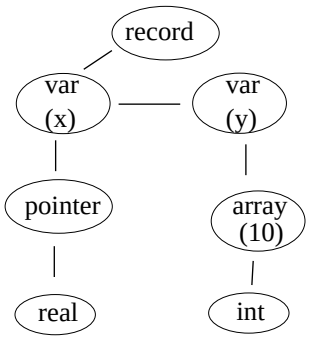
4 Las expresiones de tipo pueden contener variables cuyos valores son expresiones de tipo (para abordar el problema del polimorfismo de funciones).

Las expresiones de tipos se representan mediante árboles, con nodos interiores para los constructores de tipos y hojas para los básicos, nombres de tipos y variables de tipo.

Ejemplo de una sencilla gramática para generar expresiones de tipo:

var-decls →	var-decls **;** var-decl | var-decl
var-decl →	**id :** type-exp
type-exp →	simple-type | structured-type
simple-type →	**int** | **bool** | **real** | **char** | **void**
structured-type →	**array [num] of** type-exp |
	record var-decls **end** |
	union var-decls **end** |
	pointer to type-exp |
	proc (type-exps **)** type-exp
type-exps →	type-exps , type-exp | type-exp

```
record
    x:  pointer to real;
    y:  array [10] of int;
end
```



Las declaraciones de tipos se insertan en la tabla de símbolos, de igual modo que las declaraciones de variables.

Un **sistema de tipos**: es una serie de reglas para asignar expresiones de tipos a las distintas construcciones del lenguaje. Debe estar dotado de un mecanismo de recuperación de errores semánticos. Se dice que un lenguaje es *fuertemente tipificado* si su compilador puede garantizar que los programas que acepte se ejecutarán sin errores de tipo.

6.8.2 Comprobación e inferencia de tipos

Especificación de un comprobador de tipos sencillo: usaremos una gramática de atributos con un atributo sintetizado que el es el tipo de cada expresión.

¿ Cuando son equivalentes dos expresiones de tipos?

Equivalencia estructural: Dos expresiones son equivalentes si son el mismo tipo básico o están formadas aplicando el mismo constructor a tipos estructuralmente equivalente. Dos expresiones de tipo equivalentes tienen árboles sintácticos con la misma estructura.

Algoritmo para la comprobación de equivalencia estructural

```
function typeEqual(t1, t2 :  TypeExp) :  Boolean {
temp:Boolean;
p1,p2:TypeExp;
if (t1 and t2) son tipos simples return (t1==t2);
elseif (t1.kind==array and t2.kind==array)
      return (t1.size==t2.size and typeEqual(t1.child1,t2.child1));
elseif (t1.kind,t2.kind records o t1.kind,t2.kind uniones) {
    p1=t1.child; p2=t2.child;
    temp=true;
    while (temp and p1!=NULL and p2!=NULL) do
       if (p1.name!=p2.name) temp=false;
       elseif (!typeEqual(p1.child1, p2.child1)) temp=false;
          else { p1=p1.sibling; p2=p2.sibling; }
    return (temp and p1==NIL and p2==NILL);
}
elseif (t1.kind==pointer and t2.kind==pointer)
    return (typeEqual(t1.child1, t2.child1);
elseif (t1.kind=proc and t2.kind==proc) {
    p1=t1.child1; p2=t2.child1;
    temp=true;
    while (temp and p1!=NIL and p2!=nill) do
       if (!typeEqual(p1.child1, p2.child1)) temp=false;
       else { p1=p1.sibling; p2=p2.sibling; }
    return(temp and p1==NIL and p2==NIL and typeEqual(t1.child2,t2.child2));
    }
else return (false);
}
```

Inferencia de tipos

Obtención de los tipos de las construcciones del lenguaje a partir de las reglas semánticas que especifican cómo se combinan.

Producción	**Reglas semánticas**
Program → Decls ; Stmts	
Decls → Decls ; Decls	
Decls → **id :** Type_exp	añadetipo(id.lexema, Type_exp.tipo)
Type_exp → **char**	Type_exp.tipo= char
Type_exp → **int**	Type_exp.tipo= integer
Type_exp → Type_exp$_1$ *	Type_exp.tipo=pointer (Type_exp$_1$.tipo)
Type_exp → **array [num] of** Type_exp$_1$	Type_exp=array(num.val,Type_exp$_1$)
Exp → **num**	Exp.tipo=integer
Exp → **id**	Exp.tipo=busca(id.lexema)
Exp → Exp$_1$ **[** Exp$_2$ **]**	Exp.tipo= **if** (Exp$_2$.tipo==integer and Exp$_1$.tipo==array(s,t)) **then** t **else** error_tipo
Exp → Exp$_1$ **(** Exp$_2$ **)**	Exp.tipo= **if** (Exp$_2$.tipo==s and Exp$_1$==s '→' t) **then** t **else** error_tipo
Exp → * Exp$_1$	Exp.tipo= **if** (Exp$_1$.tipo==pointer(t)) **then** t **else** error_tipo
Stmt → **id =** Exp	Stmt.tipo= **if** (id.tipo==E.tipo) **then** vacío **else** error_tipo
Stmt → **if** Exp **then** Stmt$_1$	Stmt.tipo=**if** (Exp.tipo==boolean) **then** Stmt$_1$.tipo **else** error_tipo
Stmt → **while** Exp **do** Stmt$_1$	Stmt.tipo=**if** (Exp.tipo==boolean) **then** Stmt$_1$.tipo **else** error_tipo
Stmt → Stmt$_1$; Stmt$_2$	Stmt.tipo= **if** (Stmt$_1$.tipo==vacío and Stmt$_2$.tipo==vacío) **then** vacío **else** error_tipo

Tabla 6.1: Gramática de atributos para la comprobación e inferencia de tipos

Las sentencias carecen de tipos, pero se les puede asignar el tipo especial vacío, que permite la propagación de errores y la implementación de reglas para la recuperación.

6.8.3 Conversión de tipos

Se dice que la conversión de tipos es *ímplicita* si el compilador la realiza automáticamente (se les llama también *coerciones*). Se limitan a situaciones en donde no se pierde información (p. eje. un entero a un real). Se dice que la conversión es *explícita* si el programador debe escribir algo para motivar la conversión (un *cast).*

La conversión ímplicita se puede implementar con las correspondientes reglas semánticas. Por eje: si tenemos enteros y reales, las reglas para la conversión ímplicita de entero a real serían:

Producción	**Reglas semánticas**
Exp $\rightarrow$ **id**	Exp.tipo=busca(id.lexema)
Exp $\rightarrow$ **num_int**	Exp.tipo=integer
Exp $\rightarrow$ **num_real**	Exp.tipo=real
Exp $\rightarrow$ Exp_1 **op** Exp_2	Exp.tipo= **if** (Exp_1.tipo==integer and Exp_2.tipo==integer) **then** integer **elseif** (Exp_1.tipo==integer and Exp_2.tipo==real) **then** real **elseif** (Exp_1.tipo==real and Exp_2.tipo==integer) **then** real **elseif** (Exp_1.tipo==real and Exp_2.tipo==real) **then** real **else** error_tipo
op $\rightarrow$ + | - | *	

6.9 OTRAS COMPROBACIONES SEMÁNTICAS Y RECUPERACIÓN DE ERRORES SEMÁNTICOS

Dentro de las comprobaciones estáticas (en el momento de la compilación), tenemos la detección e información de errores como:

- Comprobaciones de tipos: *operadores aplicados a operandos incompatibles, asignación de tipos incompatibles, llamadas a funciones con tipos no adecuados, etc.*

- Comprobaciones de flujo de control: *las sentencias que hacen que el flujo de control abandone una construcción debe tener algún lugar a donde transmitir el control. Por ejemplo: un* `break` *debe estar dentro de una proposición* `while`, `for` *o* `switch` *en C.*

- Comprobaciones de unicidad: *situaciones en las que sólo se puede definir un objeto una vez exactamente. Por ej: un identificador, las etiquetas case dentro de un*
  tt switch.

Sólo nos hemos centrado en las comprobaciones de tipo. Las otras son en cierto modo rutinarias y se pueden realizar facilmente insertando acciones intercaladas en el código para realizarlas, por eje. cuando se introduce un identificador en la Tabla de Símbolos.

6.10 EJERCICIOS

1 *(0.2 ptos.) Supongamos la siguiente gramática que permite generar números enteros binarios. Diseña una gramática de atributos para convertir un número binario entero a un número decimal entero. Construye el árbol de análisis sintáctico para la cadena* 1101 *y*

$$N \rightarrow NB \mid B$$
$$B \rightarrow 1 \mid 0$$

ejecuta las reglas semánticas para encontrar su valor decimal. Indica el orden de evaluación de los atributos.

2 *(0.2 ptos.) No se sabe debido a qué extraño efecto (quizá el efecto 2000, quién sabe) los ordenadores han pasado de trabajar con cadenas de dígitos* 0 *y* 1 *a trabajar con la letra* a *para representar el* 0 *y la letra* b *para representar el* 1*. Además, se ha invertido el orden de los dígitos. Por ejemplo, la cadena* (00111) *se ha convertido en* $(bbbaa)$*. Se desea diseñar un ETDS que realice esta traducción usando la gramática base:*

$$S \rightarrow 0AS \mid 1$$
$$A \rightarrow 0SA \mid 1$$

3 *(0.25 ptos.) La siguiente gramática genera un sencillo lenguaje con expresiones aritméticas, declaración de variables e instrucciones de asignación y de entrada y salida.*

$$Programa \rightarrow Lista\_decl \;\; Lista\_sent$$
$$Lista\_decl \rightarrow Decl\,; \;\; Lista\_decl \mid \epsilon$$
$$Decl \rightarrow \textbf{var}\; id$$
$$Lista\_sent \rightarrow sent\,; \;\; Lista\_sent \mid \epsilon$$
$$Sent \rightarrow id := E \mid \textbf{leer}\; id \mid \textbf{escribir}\; id$$
$$E \rightarrow E + E \mid E \uparrow E \mid id \mid num$$

Añade las reglas semánticas y los atributos necesarios para:

- *Obtener el número de variables declaradas y una lista con el nombre de cada variable.*
- *Obtener el valor numérico de las expresiones.*
- *Construye el árbol de análisis sintáctico para la siguiente entrada indicando las dependencias entre los atributos y un posible ordenamiento de los nodos del árbol en el grafo de dependencias.*

```
var a;
var b;
leer a;
b := a ↑ a + a ;
escribir b ;
```

Suponed que el operador potencia tiene mayor precedencia que el de adición y es asociativo a la derecha.

4 *(0.2 ptos.) Diseña una gramática de atributos que calcule el tipo de las declaraciones para la siguiente gramática. Existen declaraciones de variables de tipos de datos simples (enteros y reales) y tipos de datos compuestos (vectores).*

$$
\begin{aligned}
D &\rightarrow T\ L \\
T &\rightarrow \mathbf{int} \mid \mathbf{real} \\
L &\rightarrow L\ ,\ I \mid I \\
I &\rightarrow I\ [\mathbf{num}] \mid \mathbf{id}
\end{aligned}
$$

5 *(0.2 ptos.) Dada la siguiente gramática que permite generar declaraciones de variables tipo Pascal, el atributo que indica el tipo de la variable es un atributo heredado. Transformadla para que el tipo sea ahora un atributo sintetizado. Construir el árbol de análisis sintáctico para la entrada* `x,y:char` *para ambos casos.*

$$D \rightarrow L \ : \ T$$
$$T \rightarrow \ \mathbf{integer} \mid \mathbf{char}$$
$$L \rightarrow \ L, \mathbf{id} \mid \mathbf{id}$$

6 *(0.3 ptos.) Supongamos la siguiente gramática que genera divisiones de números enteros y reales. La idea de esta gramática es que la operación de división se ha de interpretar de forma diferente dependiendo de si se trata de números enteros o reales. Por ejemplo: 5/4 puede tener el valor* 1.2 *ó* 1*, si se trata de una división en punto flotante o entera.*

$$exp \rightarrow \ exp \ / \ exp \mid \mathbf{num\_int} \mid \mathbf{num\_real}$$

Supongamos un lenguaje de programación que requiere que las operaciones en las que aparecen números enteros combinados con reales tienen que ser promocionadas a expresiones en punto flotante. Por ejemplo: la expresión $5/2/2.0$ *es* 1.25*, mientras que el significado de* $5/2/2$ *es* 1*. Se pide:*

- *Insertar las reglas semánticas necesarias para que se puedan distinguir entre estos dos tipos de operaciones. Ayuda: se necesitan tres atributos. Un atributo booleano* $isFloat$ *que indica si aparece un operando que tiene un valor en coma flotante, otro atributo* $etype$ *que indica el tipo de una subexpresión (int o real) y un atributo* val *para almacenar el valor de la expresión.*
- *Dibuja los grafos de dependencias de cada una de las producciones de la gramática. Dibuja el grafo de dependencia para la expresión* `5/2/2.0`*.*
- *¿ De qué tipo de atributos se trata?*
- *Justifica por qué es necesario dos recorridos sobre el árbol de análisis sintáctico para calcular el valor de los atributos. Una primera pasada que calcula el valor del atributo* $isFloat$*. Y otra que calcula* $etype$ *y* val*.*

- *Escribe el pseudocódigo de los procedimientos necesarios para el calculo de estos atributos. Describe las dos pasadas para el calculo de los atributos sobre el árbol generado para la entrada* `5/2/2.0`.

7 *(0.2 ptos) Considera la siguiente gramática para generar árboles binarios:*

$$btree \rightarrow \ (\ \mathbf{number}\ \ btree\ \ btree\ \ )\ \mid \mathbf{nil}$$

Diseña una gramática de atributos que compruebe si un árbol binario está ordenado, es decir, que los valores de los números del primer subárbol son menor o igual que el valor del número actual y que todos los valores del subárbol segundo son mayores o iguales que el actual. Por ejemplo, `(2 (1 nil nil) (3 nil nil))` *está ordenado y (1 (2 nil nil) (3 nil nil)) no lo está.*

8 *(0.2 ptos) Considera la siguiente gramática de atributos:*

S → A B C	B.u=S.u; A.u=B.v+C.v; S.v=A.v
A → a	A.v=2*A.u
B → b	B.v=B.u
C → c	C.v=1

(a) Dibuja el árbol de análisis sintáctico para la entrada abc, dibuja el grafo de dependencias y define un ordenamiento topológico. (b) Supón que a $S.u$ se le asigna el valor 3 antes de que comience la evaluación. ¿ Cuál es el valor de $S.v$ cuando acaba la evaluación?. (c) Considera que las ecuaciones se modifican de la forma:

S → A B C	B.u=S.u; C.u=A.v; A.u=B.v+C.v; S.v=A.v
A → a	A.v=2*A.u
B → b	B.v=B.u
C → c	C.v=C.u-2

¿ Cuál es el valor de $S.v$ después de la evaluación de los atributos si $S.u = 3$ cuando comienza la evaluación?.

9 *(0.2 ptos). En la gramática para la comprobación de tipos mostrada en la Tabla 6.1, añade las producciones y las reglas semánticas necesarias para que puedan existitr expresiones de tipo booleano (operadores* **and, or, not***) y tipo relacional (*$<, ==, >$*).*

10 *(0.2 ptos) Diseña un ETDS para traducir declaraciones de variables en C a Pascal. Usar como gramática base:*

S $\rightarrow$ T **id** L **;**
L $\rightarrow$ **,** **id** L $\mid \epsilon$
T $\rightarrow$ **int** $\mid$ **float** $\mid$ **char**

Algunos ejemplos:

Original	Traducido
int a,b,c;	var a,b,c: integer;
float d,e;	var d,e: real;

11 *(0.2 ptos) Diseña un ETDS para traducir declaraciones de funciones en C a Pascal. Usar como gramática base:*

S $\rightarrow$ Tipo_fun L **;**
Tipo_fun $\rightarrow$ **void** $\mid$ **int** $\mid$ **float**
L $\rightarrow$ F Lista_fun
Lista_fun $\rightarrow$ **,** F Lista_fun $\mid \epsilon$
F $\rightarrow$ **id (** A **)**
A $\rightarrow$ **void** $\mid$ Argu LArgu
LArgu $\rightarrow$ **,** Argu LArgu $\mid \epsilon$
Argu $\rightarrow$ Tipo **id** $\mid$ Tipo *** id**
Tipo $\rightarrow$ **int** $\mid$ **float**

Algunos ejemplos:

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;