# Tema 4
# Gramáticas Atribuidas.

Además de comprobar que un programa cumple con las reglas de la gramática , hay que comprobar que lo que se quiere hacer tiene sentido. El análisis semántico dota de un significado coherente a lo que hemos hecho en el análisis sintáctico. El chequeo semántico se encarga de que los tipos estén correctos, por ejemplo no podemos multiplicar una cadena de caracteres por un entero.

Veamos un ejemplo para tener una visión global de este tema. Supongamos que queremos hacer una pequeña calculadora que realiza las operaciones + y * . La gramática será la siguiente:

E → E +T
  | T
T → T * F
  | F
F → ( E )
  | NUM

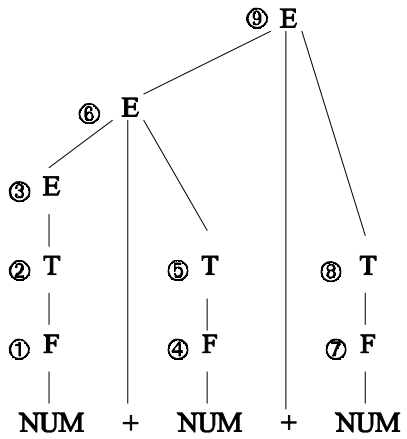
Queremos calcular: 33 + 12 + 20

Los pasos que seguiremos serán:

**1.** Análisis Léxico. (Convertimos la cadena en una secuencia de tokens con PCLEX).

33 + 12 + 20 ➪ NUM + NUM + NUM

**2.** Análisis Sintáctico




Hemos conseguido hacer el árbol sintáctico, esto quiere decir que es sintácticamente correcto.

El siguiente paso es saber que resultado nos da la operación, para ello hemos enumerado las reglas en el orden en el que se han reducido. El orden en el que van aplicando las reglas nos da el parse derecho.

# Gramáticas Atribuidas.

El hecho de que las reglas de producción esté muy cercano a la semántica que va a reconocer, podemos aprovecharlo.

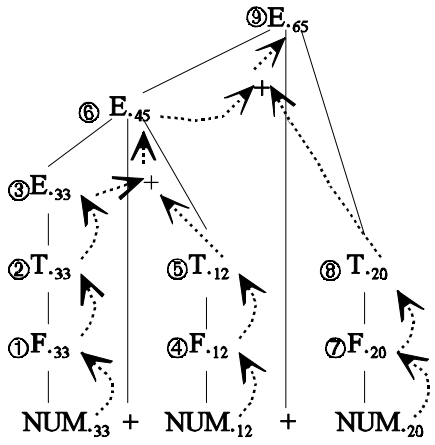
Para ello vamos a asociar unos valores a los Terminales y No Terminales de la gramática. A los Terminales le asociaremos ese valor mediante el análisis léxico, de la siguiente forma:

```
[0 - 9]+ {Convierto yytext en un entero;
          return NUM}
```

Por lo tanto lo que le llegará al analizador sintáctico es:

NUM $._{33}$ +. _ NUM $._{12}$ +._ NUM $._{20}$

Vamos a ver qué uso podría hacerse de éstos atributos para que, a medida que se hace el reconocimiento sintáctico, ir construyendo el resultado a devolver.



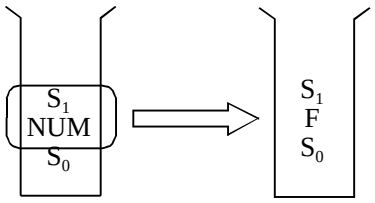

Vemos como con las asociaciones de valores y las asociaciones de acciones semánticas a las reglas de producción, podemos evaluar el comportamiento de un programa, es decir, generamos código.

ACCIONES SEMÁNTICAS

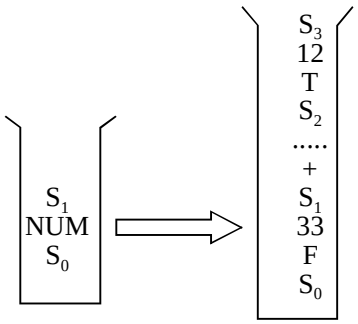
| | |
|---|---|
| $E_1 \rightarrow E_2 + T$ | $\{E_1 = E_2 + T;\}$ |
| \| T | $\{E_1 = T;\}$ |
| $T_1 \rightarrow T_2 * F$ | $\{T_1 = T_2 + F;\}$ |
| \| F | $\{T_1 = F;\}$ |
| $F \rightarrow ( E )$ | $\{F = E;\}$ |
| \| NUM | $\{F = NUM;\}$ |

Estas acciones se deben ejecutar cada vez que se reduce sintácticamente por la regla asociada.

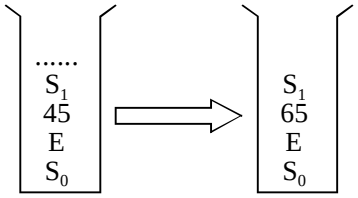
¿Cómo funciona esto realmente? A través de la pila α. En α se guardan estados, símbolos y valores, de esta forma el proceso es el siguiente




Antes solo poníamos en la pila (estados y símbolos)




Ahora añadimos los valores (estados , símbolos y valores)




De forma que no solo hemos construido el árbol sintáctico sino que además sabemos cual es el resultado de la operación que hemos realizado

El ATRIBUTO es el campo asociado a un terminal o a un no terminal.

Como ACCIÓN SEMÁNTICA no solo se puede poner una asignación o atributo, puedo además añadir código.

$E_1 \rightarrow E_2 + T$ $\{E_1 = E_2 + T; \text{printf}(E_1)\}$

Esto se ejecuta una vez por cada reducción. Si queremos que se ejecute una sola vez:

$S \rightarrow E_1$ $\{\text{printf}(E_1);\}$
$E_1 \rightarrow E_2 + T$ $\{E_1 = E_2 + T;\}$

Hay dos formas de asociar reglas semánticas con reglas de producción:

Definición dirigida por sintaxis: $A \rightarrow cdB$ $\{A = c + d\}$

Esquema de traducción: En éste es posible intercalar funciones entre el consecuente de la regla de producción. $A \rightarrow c\ d$ $\{A = c + d\}$ B. Esta regla se ejecuta en el momento en que se transita de un estado a otro.

# Gramáticas Atribuidas.

Siempre que tengamos un esquema de traducción se puede pasar a su equivalente definición dirigida por sintaxis.

Dds: $A \rightarrow cdB$ {printf(------);}
ET: $A \rightarrow cd$ { printf(-----);} $B$ ≡ $A \rightarrow cdNB$
$N \rightarrow \varepsilon$ {printf (-----);}

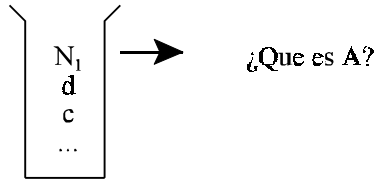
Es decir, es equivalente a introducir un No Terminal Ficticio, y ese No Terminal Ficticio es una regla ε con una acción semántica asociada.

Cada acción intermedia da lugar, a un No Terminal Ficticio distinto y a una regla ε.

Dds $A \rightarrow cdB$ {printf(c + d);} En este caso el printf sabe siempre a quien se refiere.
ET: $A \rightarrow cd$ {printf(c + d);} $B \leftrightarrow A \rightarrow cd\ N_1\ B$
$N1 \rightarrow \varepsilon$ {printf( c+ d);}
Siempre que reduzca a $N_1$ es porque antes ha aparecido cd ( por las reglas de reducción y desplazamiento).

Problemas que nos encontramos:

$A \rightarrow cd$ {A = c + d} $B \leftrightarrow$ $A \rightarrow cd\ N_1\ B$
$N1 \rightarrow \varepsilon$ {A = c + d}




Existen una serie de reglas de lo que puedo y no puedo utilizar en las acciones intermedias.
Reglas sobre atributos:

1. Un atributo solo puede ser usado (en una acción ) detrás del símbolo al que pertenece.
2. El atributo del antecedente solo se puede utilizar en la acción (del final) de su regla.
   Ej: $A \rightarrow cd\ N_1\ B$ {puedo usar el atributo de A}
   $N1 \rightarrow \varepsilon$ {No puedo usar el atributo de A}
3. En una acción intermedia solo puede hacer uso de los atributos de los simbolos que la preceden.
   Ej $A \rightarrow cd\ N_1\ B$
   En la acción de N1 solo puedo hacer uso de cd (que es lo que le precede).

Veamos todo esto en más profundidad.

# Gramáticas Atribuidas.

En este capítulo se desarrolla la traducción de lenguajes guiada por gramáticas de contexto libre. Se asocia información a una construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática que representan la construcción. Los valores de los atributos se calculan mediante “reglas semánticas” asociadas a las reglas de producción gramatical.

Hay dos notaciones para asociar reglas semánticas con reglas de producción, las definiciones dirigidas por sintaxis y los esquemas de traducción.

Las *definiciones dirigidas por sintaxis* son especificaciones de alto nivel para traducciones. No es necesario que el usuario especifique explícitamente el orden en el que tiene lugar la traducción.

Los *esquemas de traducción* indican el orden en que se deben evaluar las reglas semánticas.

Conceptualmente, tanto con las definiciones dirigidas por sintaxis como con los esquemas de traducción, se analiza sintácticamente la cadena de componentes léxicos de entrada, se construye el árbol de análisis sintáctico y después se recorre el árbol para evaluar las reglas semánticas en sus nodos. La evaluación de las reglas semánticas puede generar código, guardar información en una tabla de símbolos, emitir mensajes de error o realizar otras actividades. La traducción de la cadena de componentes léxicos es el resultado obtenido al evaluar las reglas semánticas.

No todas las implementaciones tienen que seguir al pie de la letra este esquema. Hay casos especiales de definiciones dirigidas por la sintaxis que se pueden implementar en una sola pasada evaluando las reglas semánticas durante el análisis sintáctico, sin construir explícitamente un árbol de análisis sintáctico o un grafo que muestre las dependencias entre los atributos.

# Definiciones dirigidas por sintaxis

Una *definición dirigida por sintaxis* es una gramática de contexto libre en la que cada símbolo gramatical (terminales y no terminales) tiene un conjunto de atributos asociados, dividido en dos subconjuntos llamados atributos sintetizados y atributos heredados de dicho símbolo gramatical. Si se considera un nodo de un símbolo gramatical de un árbol de análisis sintáctico como un registro con campos para guardar información, entonces un atributo corresponde al nombre de un campo.

Un atributo puede representar cualquier cosa: una cadena, un número, un tipo, una posición de memoria, etc. El valor de un atributo en un nodo de un árbol de análisis sintáctico se define mediante una regla semántica asociada a la regla de producción utilizada en dicho nodo. El valor de un *atributo sintetizado* de un nodo se calcula a partir de los valores de los atributos hijos de dicho nodo en el árbol de análisis sintáctico; el valor de un *atributo heredado* se calcula a partir de los valores de los atributos en los hermanos y el padre de dicho nodo.

Las *reglas semánticas* establecen las dependencias entre los atributos que serán representadas mediante un grafo. Del grafo de dependencias se obtiene un orden de evaluación de las reglas semánticas. La evaluación de las reglas semánticas define los valores de los atributos en los nodos del árbol de análisis sintáctico para la cadena de entrada. Una regla semántica también puede tener efectos colaterales, por ejemplo, imprimir un valor o actualizar una variable global. Por supuesto, una aplicación no necesita construir explícitamente un árbol de análisis sintáctico o un grafo de dependencias; sólo tiene que producir el mismo resultado para cada cadena de entrada.

Un árbol de análisis sintáctico que muestre los valores de los atributos en cada nodo se denomina un árbol de análisis sintáctico con anotaciones. El proceso de calcular los valores de los atributos en los nodos se denomina *anotar* o *decorar* el árbol de análisis sintáctico.

## ✯ Forma de una definición dirigida por sintaxis

En una definición dirigida por sintaxis, cada regla de producción $A \rightarrow \alpha$ tiene asociado un conjunto de reglas semánticas de la forma $b:=f(c_1, c_2, ..., c_k)$, donde $f$ es una función, y, o bien

**1.** $b$ es un atributo sintetizado de A y $c_1, c_2, ..., c_k$ son atributos de los símbolos de $\alpha$, o bien

**2.** $b$ es un atributo heredado de uno de los símbolos de $\alpha$ , y $c_1, c_2, ..., c_k$ son atributos de los restantes símbolos de $\alpha$ o bien de A.

En cualquier caso, se dice que el atributo $b$ depende de los atributos $c_1, c_2, ..., c_k$.
Una *gramática con atributos* es una definición dirigida por sintaxis en la que las funciones en las reglas semánticas no pueden tener efectos colaterales.

Las funciones de las reglas semánticas a menudo se escribirán como expresiones. Ocasionalmente el único propósito de una regla semántica en una definición dirigida por sintaxis es crear un efecto colateral. Dichas reglas semánticas se escriben como llamadas a procedimientos o fragmentos de programa. Se pueden considerar como reglas que definen los valores de atributos sintetizados ficticios del no terminal del lado izquierdo de la regla de producción asociada; no se muestran el atributo ficticio y el signo := de la regla semántica.

Las reglas semánticas se ejecuta en el momento en que se reduce por su regla asociada.

**Ejemplo 4.1.** La definición dirigida por sintaxis de la figura 4.2 es para un programa que implementa una calculadora. Esta definición asocia un atributo sintetizado con un valor entero llamado *val* a cada uno de los no terminales *E, T y F*. Para cada regla de producción de *E, T y F*, la regla semántica calcula el valor del atributo *val* para el no terminal del lado izquierdo a partir de los valores de los no terminales del lado derecho.

| PRODUCCIÓN | REGLAS SEMÁNTICAS |
|---|---|
| L → E '\n' | print (E.val) |
| E → $E_1$ + T | E.val = E1.val + T.val |
| E → T | E.val = T.val |
| T → $T_1$ * F | T.val = T1.val * F.val |
| T → F | T.val = F.val |
| F → ( E ) | F.val = E.val |
| F → DIGITO | F.val = DIGITO.valex |

**Figura 4.2** .Definición dirigida por sintaxis de una calculadora sencilla

El componente léxico DIGITO tiene un atributo sintetizado *valex* cuyo valor viene proporcionado por el analizador léxico. La regla asociada a la producción *L → E '\n'* para el no terminal inicial *L* es sólo un procedimiento que imprime como resultado el valor de la expresión aritmética generada por *E*; se puede considerar que esta regla define un falso atributo para el no terminal *L*. Más adelante veremos una especificación en YACC para esta calculadora, para ilustrar la traducción durante el análisis sintáctico LALR(1).

En una definición dirigida por sintaxis, se asume que los terminales sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla semántica para los terminales. El analizador léxico es el que proporciona generalmente los valores de los atributos de los terminales. Además se asume que el símbolo inicial no tiene ningún atributo heredado, a menos que se indique lo contrario.

### ✯ Atributos sintetizados

Los atributos sintetizados son muy utilizados en la práctica. Una definición dirigida por sintaxis que usa atributos sintetizados exclusivamente se denomina *definición con atributos sintetizados*. Siempre se puede anotar un árbol de análisis sintáctico para una definición con atributos sintetizados mediante la evaluación de las reglas semánticas para los atributos en cada nodo de forma ascendente, de las hojas a la raíz.

**Ejemplo 4.2.** La definición con atributos sintetizados del ejemplo 4.1 especifica una calculadora que lee una línea de entrada que contiene una expresión aritmética que incluye dígitos, paréntesis, los operadores + y *, seguida de un carácter de nueva línea '\n', e imprime el valor de la expresión. Por ejemplo, dada la expresión 3*5+4 seguida de una nueva línea, el programa imprime el valor 19. La figura 4.3 contiene un árbol de análisis sintáctico con anotaciones para la entrada 3*5+4\n. El resultado, que se imprime en la raíz del árbol, es el valor de *E.val* en el primer hijo de la raíz.

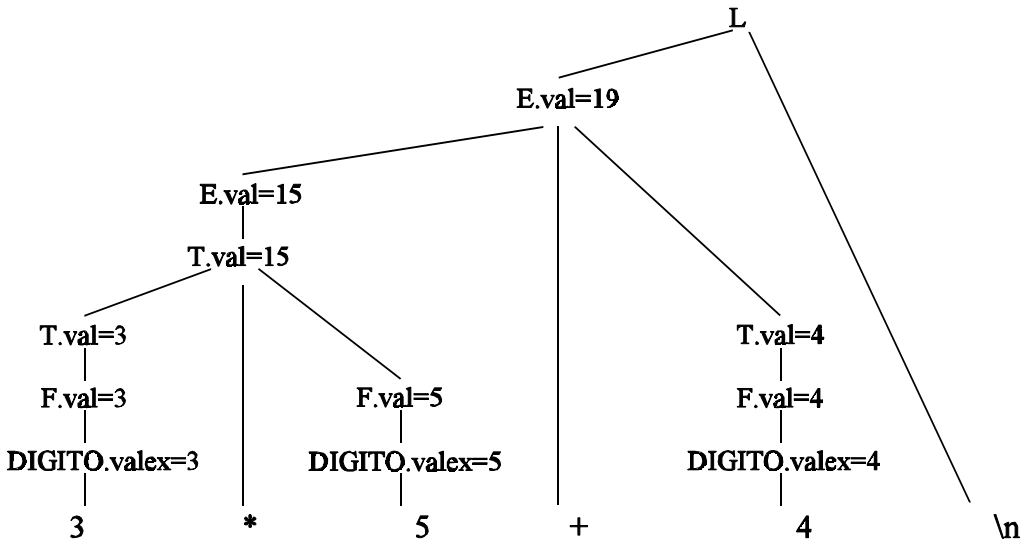



**Figura 4.3.** Arbol de análisis sintáctico con anotaciones para 3*5+4\n

Para ver cómo se calculan los valores de los atributos, considérese el nodo situado en el extremo de la izquierda, que corresponde al uso de la producción $F \rightarrow DIGITO$. La regla semántica correspondiente, *F.val:= DIGITO.valex*, establece que el atributo *F.val* en el nodo tiene el valor 3 porque el valor de *DIGITO.valex* en el hijo de este nodo es 3. De forma similar, en el padre de este nodo *F*, el atributo *T.val* tiene el valor 3.

A continuación considérese el nodo para la producción $T \rightarrow T * F$. El valor del atributo *T.val* en este nodo está definido por:

| PRODUCCIÓN | REGLA SEMÁNTICA |
|---|---|
| $T \rightarrow T_1 * F$ | $T.val := T_1.val * F.val$ |

Cuando se aplica la regla semántica en este nodo, $T_1.val$ tiene el valor 3 del hijo izquierdo y *F.val* el valor 5 del hijo derecho. Por tanto, *T.val* adquiere el valor 15 en este nodo.

La regla asociada con la producción para el no terminal inicial $L \rightarrow E$ *\n* imprime el valor de la expresión generada por *E*.

## ✯ Atributos heredados

Un atributo heredado es uno cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos en el padre y/o de los hermanos de dicho nodo. Los atributos heredados sirven para expresar la dependencia de una construcción de un lenguaje de programación en el contexto en el que aparece. Por ejemplo, se puede utilizar un atributo heredado para comprobar si un identificador aparece en el lado izquierdo o en el derecho de una asignación para decidir si se necesita la dirección o el valor del identificador. Aunque siempre es posible reescribir una definición dirigida por sintaxis para que sólo se utilicen atributos sintetizados, a veces es más natural utilizar definiciones dirigidas por la sintaxis con atributos heredados.

En el siguiente ejemplo, un atributo heredado distribuye la información sobre los tipos a los distintos identificadores de una declaración.

**Ejemplo 4.3**. Una declaración generada por el terminal *D* en la definición dirigida por sintaxis en la figura 4.4 consta de la palabra clave *INT* o *REAL*, seguida de una lista de identificadores.

| PRODUCCIÓN | REGLAS SEMÁNTICAS |
|---|---|
| D → T L | L.her:= T.tipo |
| T → INT | T.tipo := integer |
| T → REAL | T.tipo := real |
| L → $L_1$, ID | $L_1$.her := L.her<br>añadetipo(ID.ptr_tds, L.her) |
| L → ID | añadetipo(ID.ptr_tds, L.her) |

**Figura 4.4** Definición dirigida por sintaxis con el atributo heredado *L.her*

El no terminal *T* tiene un atributo sintetizado *tipo*, cuyo valor viene determinado por la palabra clave de la declaración. La regla semántica *L.her := T.tipo*, asociada con la regla de producción *D → T L*, asigna al atributo heredado *L.her* el tipo de la declaración. Entonces las reglas pasan este tipo por el árbol de análisis sintáctico utilizando el atributo heredado *L.her*. Las reglas asociadas con las producciones de *L* llaman al procedimiento *añadetipo* para añadir el tipo de cada identificador a su entrada en la tabla de símbolos (apuntada por el atributo *ptr_tds*).

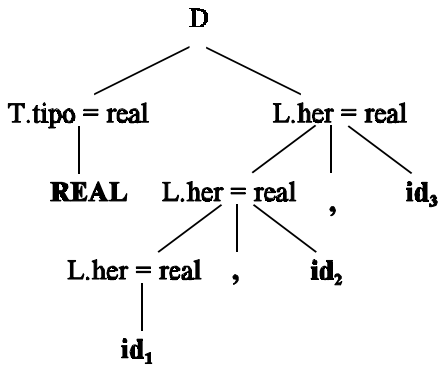



**Figura 4.5** Arbol de análisis sintáctico con el atributo heredado *her* en cada nodo etiquetado *L*

En la figura 4.5 se muestra un árbol de análisis sintáctico con anotaciones para la frase ***real*** $\boldsymbol{id_1, id_2, id_3}$. El valor de *L.her* en los tres nodos de *L* da el tipo de los identificadores $\boldsymbol{id_1, id_2, id_3}$. Estos valores se determinan calculando el valor del atributo *T.tipo* en el hijo izquierdo de la raíz y evaluando después *L.her* de forma descendente en los tres nodos de *L* en el subárbol derecho de la raíz. En cada nodo de *L* también se llama al procedimiento *añadetipo* para insertar en la tabla de símbolos el hecho de que el identificador en el hijo derecho de este nodo tiene tipo real.

### ✯ 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 dirigido llamado *grafo de dependencias*.

Antes de construir un grafo de dependencias para un árbol de análisis sintáctico, se escribe cada regla semántica en la forma $b:=f(c_1, c_2, ..., c_k)$, introduciendo un falso atributo sintetizado *b* para cada regla semántica que conste de una llamada de procedimiento. El grafo tiene un nodo por cada atributo y una arista al nodo de *b* desde el nodo de *c* si el atributo *b* depende del atributo *c*. Más detalladamente, el grafo de dependencias para un determinado árbol de análisis sintáctico se construye de la siguiente manera:

**for** cada nodo *n* en el árbol de análisis sintáctico **do**
  **for** cada atributo *a* del símbolo gramatical en el nodo *n* **do**
    construir un nodo en el grafo de dependencias para *a*;
**for** cada nodo *n* en el árbol de análisis sintáctico **do**
  **for** cada regla semántica $b:=f(c_1, c_2, ..., c_k)$
    asociada con la producción utilizada en *n* **do**
    **for** *i*:= 1 **to** *k* **do**
      construir una arista desde el nodo para $c_i$ hasta el nodo para *b*;

Por ejemplo, supóngase que *A.a := f(X.x, Y.y)* es una regla semántica para la producción $A \rightarrow XY$. Esta regla define un atributo sintetizado *A.a* que depende de los atributos *X.x* y *Y.y*. Si se utiliza esta producción en el árbol de análisis sintáctico, entonces habrá tres nodos, *A.a*, *X.x* e *Y.y*, en el grafo de dependencias con una arista desde *X.x* hasta *A.a* puesto que *A.a* depende de *X.x* y una arista desde *Y.y* hasta *A.a* puesto que *A.a* también depende de *Y.y*.
Si la regla de producción $A \rightarrow XY$ tiene asociada la regla semántica *X.i := g(A.a, Y.y)*, entonces habrá una arista desde *A.a* hasta *X.i* y también una arista desde *Y.y* hasta *X.i*, puesto que *X.i* depende tanto de *A.a* como de *Y.y*.

**Ejemplo 4.4.** Siempre que se utilice la siguiente regla de producción en un árbol de análisis sintáctico, se añaden al grafo de dependencias las aristas que se muestran en la figura 4.6.

| PRODUCCIÓN | REGLA SEMÁNTICA |
|---|---|
| $E \rightarrow E_1 + E_2$ | $E.val := E_1.val + E_2.val$ |

Los tres nodos del grafo de dependencias marcados con ● representan los atributos sintetizados *E.val*, $E_1.val$ y $E_2.val$ en los nodos correspondientes del árbol de análisis sintáctico.

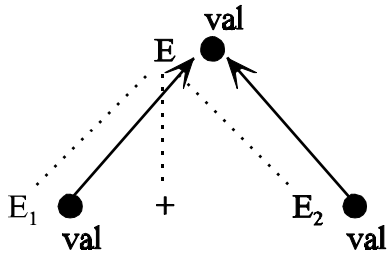



**Figura 7.6**. *E.val* se sintetiza a partir de $E_1.val$ y $E_2.val$

La arista hacia *E.val* desde $E_1.val$ muestra que *E.val* depende de $E_1.val$ y la arista hacia *E.val* desde $E_2.val$ muestra que *E.val* también depende de $E_2.val$. Las líneas con puntos representan al árbol de análisis sintáctico y no son parte del grafo de dependencias.

**Ejemplo 4-5.** En la figura 4.7 se muestra el grafo de dependencias para el árbol de análisis sintáctico de la figura 4.5. Los nodos en los grafos de dependencias están marcados con números; estos números serán utilizados posteriormente. Hay una arista ( ② ) hacia el nodo *L.her* desde el nodo *T.tipo* porque el atributo heredado *L.her* depende del atributo *T.tipo* según la regla semántica *L.her:=T.tipo* para la regla de producción $D \rightarrow T\ L$. Las dos aristas que apuntan hacia abajo ( ④ y ⑥) surgen porque depende de *L.her* según la regla semántica $L_1.her := L.her$ para la regla de producción $L \rightarrow L_1$, *ID*. Cada una de las reglas semánticas *añadetipo*(id.ptr_tds, *L.her*) asociada con las producciones de *L* conduce a la creación de un falso atributo. Los nodos * se construyen para dichos falsos atributos (aristas ③, ⑤ y ⑦).

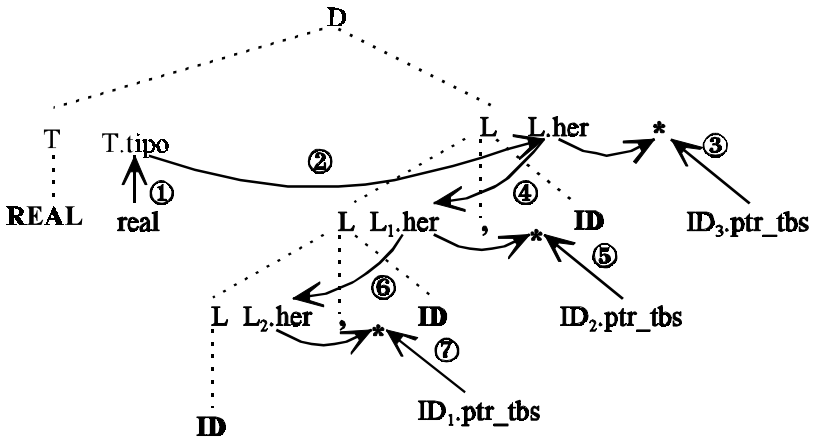



**Figura 4.7** Grafo de dependencias para el árbol de análisis sintáctico de la figura 4.5

## ✯ Orden de evaluación

Un ordenamiento topológico de un grafo dirigido acíclico es todo ordenamiento $m_1$, $m_2$, ..., $m_k$ de los nodos del grafo tal que las aristas vayan 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.

Todo ordenamiento topológico de un grafo de dependencias da un orden válido en el que se pueden evaluar las reglas semánticas asociadas con los nodos de un árbol de análisis sintáctico. Es decir, en el ordenamiento topológico, los atributos dependientes $c_1$, $c_2$,..., $c_k$ en una regla semánticas $b:=f(c_1, c_2,..., c_k)$ están disponibles en un nodo antes de que se evalúe $f$.

La traducción especificada por una definición dirigida por sintaxis se puede precisar como sigue. Se utiliza la gramática subyacente para construir un árbol de análisis sintáctico para la entrada. El grafo de dependencias se construye como se indica más arriba. A partir de un ordenamiento topológico del grafo de dependencias, se obtiene un orden de evaluación para las reglas semánticas. La evaluación de las reglas semánticas en este orden produce la traducción de la cadena de entrada.

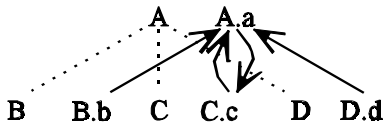
**Ejemplo 4.6.** Cada una de las aristas en el grafo de dependencias de la figura 4.7 van numeradas por tanto, si hacemos un ordenamiento topológico del grafo de dependencias se obtiene el siguiente programa

① *T.tipo = real*
② *L.her = T.tipo*
③ *añadetipo($ID_3$.ptr_tbs, L.her);*
④ *$L_1$.her = L.her*
⑤ *añadetipo($ID_2$.ptr_tbs, L.her);*
⑥ *$L_2$.her = $L_1$.her*
⑦ *añadetipo($ID_1$.ptr_tbs, L.her);*

La evaluación de estas reglas semánticas almacena el tipo real en la entrada de la tabla de símbolos para cada identificador.

Se dice que una gramática atribuida es *circular* si el grafo de dependencias para un árbol sintáctico generado por la gramática tiene un ciclo.

A→ B C D  A.a = f (B.b, C.c, D.d)
C.c = g (A.a)




No se consigue encontrar un orden de evaluación sólo si el grafo de dependencias tiene un ciclo.

✯ **Gramática L-atribuida** ( o atribuida por la izquierda, o definición con atributos por la izquierda). En toda regla $A \rightarrow X_1 X_2 ... X_n$, cada atributo heredado de $X_j$ $1 \leq j \leq n$ depende sólo de:

**1.** Los atributos (sintetizados o heredados) de los símbolos $X_1 X_2 ... X_{j-1}$.
**2.** Los atributos heredados de A.

Este tipo de gramáticas no producen ciclos y además admiten un orden de evaluación en profundidad.

```
visitaprof (nodo n)
{  for cada hijo m de n, de izquierda a derecha
        {
          evaluar los atributos heredados de m;
          visitaprof(m)
        }
   evaluar los atributos sintetizados de n;
}
```

**Ejemplo 4.7** de gramática L-atribuida donde (D - declaración ; T - tipo; L - Lista de identificadores)

| | |
|---|---|
| D → T L | {L.tipo = T.tipo;} |
| T → INTEGER | {T.tipo = INTEGER;} |
| T → REAL | {T.tipo = REAL;} |
| L → L', ID | {L'.tipo = L.tipo;<br>ID.tipo = L.tipo;} |
| L → ID | {ID.tipo = L.tipo;} |

**Ejemplo 4.8** de gramática no L-atribuida

| | |
|---|---|
| D → ID T | {ID.tipo = T.tipo;<br>D.tipo = T.tipo;} |
| D → ID D' | {ID.tipo = D'.tipo;<br>D.tipo = D'.tipo;} |

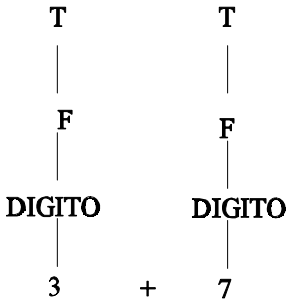
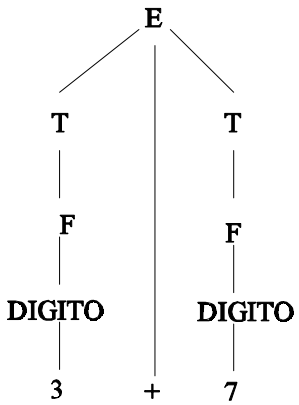
- **Gramática S-atribuida**

Es una gramática atribuida que sólo contiene atributos sintetizados Una gramática S-atribuida es también L-atribuida.

Los atributos sintetizados se pueden evaluar con un analizador sintáctico ascendente conforme la entrada es analizada. El analizador sintáctico puede conservar en su pila los valores de los atributos sintetizados asociados con los símbolos gramaticales. Siempre que se haga una reducción se calculan los valores de los nuevos atributos sintetizados a partir de los atributos que aparecen en la pila para los símbolos gramaticales del lado derecho de la producción con la que se reduce.

| | |
|---|---|
| L → E '\n' | print(E.val) |
| E → $E_1$ + T | E.val = $E_1$.val + T.val |
| E → T | E.val = T.val |
| T → $T_1$ * F | T.val = $T_1$.val * F.val |
| T → F | T.val = F.val |
| F → ( E ) | F.val = E.val |
| F → DIGITO | F.val = DIGITO.valex |

**Ejemplo 4.9** Analizaremos la entrada **3 + 7 '\n'**

| Árbol | Pila | |
|---|---|---|
|   | DIGITO | DIGITO.valex = 3 |
|   | F | (F.val = DIGITO.valex)<br>(F.val = $1)<br><br>F.val = 3 |
|   | T<br>+<br>E | T.val = 7 ⟵ $1<br>---- ⟵ $2<br>E.val = 3 ⟵ $3 |
|   | E | **(E.val = E1.val + T.val**<br>**E.val = $1 + $3)**<br><br>**E.val = 10** |

Ahora, la pila además de tener estados y símbolos, también tiene atributos.

# Esquemas de traducción

Un *esquema de traducción* es una gramática de contexto libre en la que se encuentran intercalados, en el lado derecho de la regla de producción, fragmentos de programas a los que hemos llamado acciones semánticas. Un esquema de traducción es como una definición dirigida por sintaxis, con la excepción de que el orden de evaluación de las reglas semánticas se muestra explícitamente. La posición en la que se ejecuta alguna acción se da entre llaves y se escribe en el lado derecho de una producción por ejemplo.

A → cd {printf(c + d)} B

Cuando se diseña un esquema de traducción, se deben respetar algunas limitaciones para asegurarse de que el valor de un atributo esté disponible cuando una acción se refiera a él. Estas limitaciones, motivadas por las definiciones dirigidas por sintaxis, garantizan que las acciones no hagan referencia a un atributo que aún no haya sido calculado.

El ejemplo más sencillo ocurre cuando sólo se necesitan atributos sintetizados. En este caso, se puede construir el esquema de traducción creando una acción que conste de una asignación para cada regla semántica y colocando esta acción al final del lado derecho de la regla de producción asociada. Por ejemplo, la producción y la regla semántica

| Producción | Regla Semántica |
|---|---|
| $T \rightarrow T_1 * F$ | $T.val := T_1.val * F.val$ |

dan como resultado la siguiente producción y acción semántica:

$T \rightarrow T_1 * F \{T.val := T_1.val * F.val\}$

Si se tienen atributos tanto heredados como sintetizados, se debe ser más prudente:

1. Un atributo heredado para un símbolo en el lado derecho de una regla de producción se debe calcular en una acción antes que dicho símbolo.
2. Una acción no debe referirse a un atributo sintetizado de un símbolo que esté a la derecha de la acción.
3. 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 hace referencia. La acción que calcula dichos atributos se puede colocar generalmente al final del lado derecho de la producción.

A partir de una gramática L-atribuida es posible construir un esquema de traducción equivalente bien definido

D → T {L.her = T.tipo} L;
T → INT {T.tipo = integer}
T → REAL { T.tipo = real}
L → {$L_1$.her = L.her} L1, ID {añadetipo (ID.ptr_tbs, L.her)}
L → ID {añadetipo(ID.ptr_tbs, L.her)}

# Análisis LALR con atributos

Vamos a ver como funciona el análisis LALR con atributos.

Recordemos que el LALR(1) es capaz de saber en todo momento qué regla aplicar sin margen de error. Este hecho es el que aprovechamos para pasar a ejecutar las acciones de forma segura. En el caso de retroceso, es posible que tras ejecutar una acción sea necesario retroceder porque nos hayamos dado cuenta de que esa regla no era la correcta. En tal caso, ¿cómo deshacer una acción?. Como ello es imposible, en un análisis con retroceso no es posible ejecutar acciones a la vez que se hace el análisis, sino que habría que reconocer primero la sentencia, y en una fase posterior ejecutar las acciones con seguridad.

Hay esquemas de traducción que no pueden ser reconocidos por analizadores LALR(1). Ejemplo: Dado el esquema de traducción de la izquierda lo pasamos a la definición dirigida por sintaxis de la derecha (Añadiendole un No Terminal Ficticio que son reglas de producción ε).

$$
\begin{array}{l|c|l}
S \rightarrow A & & S \rightarrow A \\
S \rightarrow B & & S \rightarrow B \\
B \rightarrow ab\{acción_1\}\ cd & \Longrightarrow & B \rightarrow ab\ V\ cd \\
A \rightarrow ab\ \{acción_2\}cde & & A \rightarrow ab\ Ucde \\
 & & V \rightarrow \{acción_1\} \\
 & & U \rightarrow \{acción_2\}
\end{array}
$$

En este caso se produce un conflicto Reduce/Reduce, porque la gramática no es LALR.

Un conflicto Reduce/Reduce se produce cuando llega un momento del análisis en el que no se sabe que regla aplicar. En el caso anterior, si leemos

a b c d $,
↑

cuando estamos en el punto marcado, no podemos saber que regla ε aplicar, si U o V. Para solucionar este problema necesitaríamos un análisis LALR(3), ya que no es suficiente ver la ‘c’, tampoco es suficiente ver la ‘d’, es necesario llegar un carácter más allá, a la ‘e’ o al ‘$’ para saber en que regla estamos, y, por tanto , que acción aplicar.

Hay casos más liosos, que se producen en gramáticas que son ambiguas, o sea, una sentencia puede tener más de un árbol sintáctico. Tal es el caso de

S → <if> <cond> <then> S
S → <if> <cond> <then> S <else> S
S → <id> < := > <exp>

en la que se producen conflictos shift/Reduce.

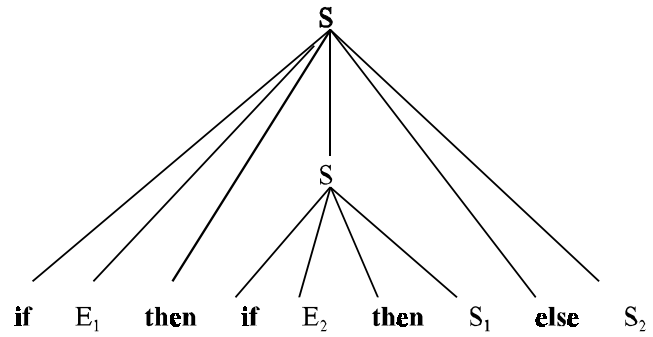
Este conflicto ocurre también en gramáticas como:

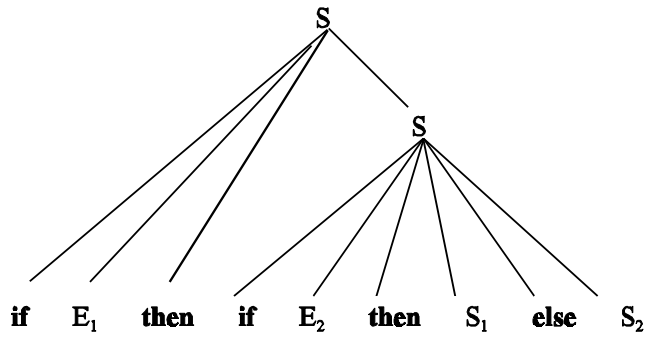
S → aBadd
S →aCd
B → a
C→ aa

Al leer a a a d $
↑ no se sabe si Reducir o Desplazar.

Podemos aplicar reglas desambiguantes (si la gramática es ambigua), que lo que hacen es indicar de forma implícita si reducir o desplazar.

Por ejemplo la cadena: *if E1 then if E2 then S1 else S2* tiene dos posibles árboles sintácticos




En todos los lenguajes de programación con proposiciones condicionales de esta forma, se refiere al segundo árbol de análisis sintáctico. La regla general es, "emparejar cada **else** con el **then** sin emparejar anterior más cercano". Esta regla para eliminar ambigüedades se puede incorporar directamente a la gramática. La gramática se puede reescribir de la siguiente forma:

S → S_completo
| S_incompleto

S_completo → <if> Expr <then> S_completo <else> S_completo
| <id> < := > Expr

S_incompleto → <if> Expr <then> S
|<if> Expr <then> S_completo <else> S_incompleto

que asocia siempre el *else* al *if* más inmediato.

De forma que un S_completo es o una proposición **if-then-else** que no contenga proposiciones incompleta o cualquier otra clase de proposición no condicional. Es decir, dentro de un **if** con **else** solo se permiten if completos.

La recursividad derecha hace aumentar la pila, porque primero efectúa todos los desplazamientos, y por último las reducciones, con lo que la pila aumenta mucho. Si empleamos recursividad a izquierda., las reducciones se intercalan con los desplazamientos.

Ejemplo: (lista de identificadores)

| | |
|---|---|
| L → id ; | L' → id |
| L → id, L | L' → L', id |
| | L → L'; |

En un esquema atribuido, lo que hacemos es que en la pila de estados y terminales y no terminales, asociamos a cada $X \in \{T \cup N\}$ sus atributos, de manera que podemos acceder en cada momento a los atributos de la pila que corresponden a la regla actual.

# El generador de analizadores sintáctico: PCYACC

## ✯ El Metacompilador YACC

YACC es un metacompilador, que acepta como entrada una gramática de contexto libre, y genera el autómata finito que reconoce tal lenguaje. En concreto, no acepta todas las gramáticas pero sí un conjunto bastante extenso de ellas: las LALR. Además permite asociar acciones al reconocimiento de las reglas, lo que nos permite efectuar operaciones dirigidas por sintaxis. Permite igualmente el asignar un atributo a cada elemento de la gramática ( terminal o no terminal), lo cual facilita las labores de interpretación.

YACC actúa en base a unos tokens (símbolos terminales), que deben ser generados por el analizador lexicográfico. Este analizador léxico podemos hacerlo a mano, o bien a través del código generado por LEX.

## ✯ El formato de un programa YACC:

Un programa fuente en YACC tiene la siguiente sintaxis:

```
Area de definiciones
%%
Area de reglas
%%
Area de funciones
```

Para ilustrar un programa fuente en YACC, partiremos de la siguiente gramática para expresiones aritmética:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow ( E ) \mid ID \mid NUM$

**El área de definiciones.** Hay dos secciones opcionales en la parte de declaraciones de un programa en YACC. En la primera sección, se ponen declaraciones ordinarias en C, delimitadas por %{ y %}. Aquí se sitúan las declaraciones de todas las variables temporales usadas en el área de reglas o los procedimientos de la segunda y tercera secciones.

De esta forma, el usuario también puede definir variables globales al igual que en LEX, colocándolas al comienzo en la parte de declaraciones, entre %{ y %}.

Además en el área de definiciones declararemos los componentes léxicos (tokens) usados en la gramática.

```
%token ID NUM
```

Los terminales se pueden expresar de dos formas:

- Si es un solo carácter, y el analizador léxico lo devuelve tal cual (su código ASCII), se deben poner entre comillas simples (notación C). Ejemplo: '+'

- Si es una cadena de caracteres devuelta por el analizador léxico, se pone la palabra que lo representa, y debe ser declarada en la parte de declaraciones.
  En nuestro ejemplo, la proposición
  %token ID NUM

declara que ID y NUM son componentes léxicos (token).

Por convención, los símbolos terminales (token) se escriben con mayúscula. Internamente cada token se representa por un entero, empezando desde el 257 (ya que los 255 anteriores son utilizados para los caracteres ASCII, y el 256 es utilizado como un token de error). Por lo tanto la proposición
%token ID NUM

será equivalente a
*# define ID nº$_1$*
*# define NUM nº$_2$*

YACC se asegura de que no hay dos tokens con el mismo número.

Los tokens declarados en esta sección se pueden utilizar después en el área de reglas y de funciones.

Nosotros trabajaremos con LEX y YACC. Veamos que ocurre cuando se produce un error del tipo "caracteres no permitidos".

```
%%    /*OPCIÓN A*/
[0 - 9]+              {return NUM;}
[A- Z][A - Z0 -9]*    {return ID;}
[" "\t\n]             {; }
.                     {return yytext[0];}    /*error sintáctico. Le pasa el error
                                              a YACC*/

%%    /*OPCIÓN B*
[0 - 9]+              {return NUM;}
[A- Z][A - Z0 -9]*    {return ID;}
[" "\t\n]             {; }
'('|')'|'*'|'+'       {return yytext[0];}    /*aquí le pasa una cadena de
                                              caracteres*/
.                     {printf ("Error léxico");}
```

Nosotros utilizaremos el método del mínimo esfuerzo, es decir la opción A (le pasaremos el error al YACC).

En LEX ya no hay que poner el main, porque el que tiene el control es el YACC. Ahora en la zona de funciones del YACC habrá que poner un main con una llamada al analizador sintáctico.

```
%token ID NUM
%%
.....
%%
void main ( ){
        yyparse( );
}
```

**El área de reglas** es la más importante, y en la que se especifican las reglas que forman la gramática que queremos reconocer.

En la parte de especificación en YACC después del primer par %% se ponen las reglas. Una regla gramatical tienen la forma:

```
No_terminal: Consecuente;
```

donde consecuente representa una secuencia de cero o más terminales y no terminales.

El conjunto de reglas de producciones de nuestro ejemplo se escribiría en YACC como:

```
                          %%    /* área de reglas */
E → E + T                 e : e '+' t
    | T                     | t
                            ;
T → T * F                 t : t '*' f
    | F                     | f
                            ;
F → ( E )                 f : '(' e ')'
    | ID                    | ID
    | NUM                   | NUM
                            ;
```

donde , por convención, los símbolos no terminales se escriben en minúscula y los terminales en mayúscula. Las flechas se sustituyen por '**:**'. Además cada regla debe acabar en '**;**'

Junto con los terminales y no terminales del consecuente, se pueden expresar acciones. Ejemplo:

```
E : E '+' T      {printf("Esto es una suma");};
```

Si hay varias reglas con el mismo antecedente, es conveniente agruparlas, y se pueden separar por una barra vertical ( | ) sin necesidad de punto y coma, ni de volver a poner el antecedente. Por lo tanto las reglas gramaticales:

```
E  : E '+' T {printf ("Esto es una suma"):};
E  : T       {printf("Esto es un término");};
```

se le pueden dar a YACC como

```
e  : e '+' t {printf ("Esto es una suma"):}
   | t       {printf("Esto es un término");}
   ;
```

Como ya hemos dicho, en una regla se pueden indicar acciones, que no es más que una secuencia de proposiciones en C. Pero la verdadera potencia del YACC es que a cada terminal o no terminal de una regla se le puede asociar un atributo. Los nombres de atributos son siempre los mismos y dependen de la posición que el terminal o no terminal ocupa dentro de la regla y no del terminal o no terminal en sí. De forma que el símbolo $$ se refiere al antecedente de la regla, mientras que $i se refiere al valor asociado con el i-ésimo símbolo gramatical del consecuente, es decir

el antecedente es $$
el 1er símbolo del consecuente es $1.
el 2º símbolo del consecuente es $2.

La acción semántica se realiza siempre que se reduzca por la producción asociada, por lo que normalmente la acción semántica calcula un valor para $$ en función de los $i.

Ejemplo: Suponemos que el atributo de los terminales y no terminales es de tipo entero. Se tendría:

```
%%
e : e '+' t      {$$ = $1 + $3;}
  | t            {$$ = $1;}
  ;
t  : t '*' f     {$$ = $1 * $3;}
  | f            {$$ = $1;}
  ;
f  : '(' e ')'   {$$ = $2;}
   | ID          {$$ = $1;}
   ;
```

Observar, en la última regla, $1 se refiere al valor asociado a ID, pero ID es un terminal, ¿qué valor tiene?. Es el analizador léxico. El que se tiene que encargar de asignarle un valor. Recordemos que en LEX había una variable global yylval que permitía asociar información adicional. Pues bien, YACC utiliza esta misma variable para asociar un valor a un token. Mediante esta variable LEX se comunica con YACC. ¿Qué ocurre cuando le pasamos una secuencia de entrada a LEX?

```
  27   +    13    +   25
NUM '+' NUM '+' NUM
```

LEX debe cargar los valores de los atributos de los tokens, y debe de hacerlo antes de retornar el token (antes de hacer el *return NUM;*)

```
%%
[0-9]+  {yylval = atoi (yytext);
         return NUM;}
```

atoi convierte de texto a int, ya que se supone que el valor del atributo de NUM es entero, por lo tanto la variable yylval es del mismo tipo, dicha variable pone el valor del atributo del token que le devuelve.

El axioma inicial se debe poner en la zona de declaraciones de la forma:
*%start nombre_axioma*.

Por defecto, el axioma inicial se considera el antecedente de la primera regla gramatical que aparece en el área de reglas.

También es posible colocar acciones en medio de una regla. En tal caso, dicha acción equivale a un no terminal Ni que tiene una regla de la forma : Ni → ε.

```
A : B {$$ = 1} C {x = $2; y = $3;}
```

equivale a

```
Ni : ε {$$ = 1;};
A : B Ni C {x = $2; y = $3;};
```

Hemos utilizado alegremente las variables que mantienen los valores de los atributos $$, $1, $2, etc en las acciones semánticas en YACC sin hacer ninguna consideración acerca de sus tipos.

## El generador de analizadores sintáctico: PCYACC

Yacc necesita una gestión de atributos, es decir no solo hay que decirle a que terminal o no terminal está asociado, sino que también tenemos que decirle de que tipo es el atributo. Para ello existe una macro YYSTYPE que indica de que tipo son todos los atributos.

```
%{
#define YYSTYPE int
%}
%token ID NUM
%%
```

YYSTYPE se pone en el área de definiciones, es una definición global. En el ejemplo anterior indica que todos los atributos son enteros. En PCYACC, es obligatorio poner YYSTYPE, aunque no utilicemos atributos para nada : PCYACC siempre tiene que saber los tipos de los atributos.

Ahora bien ¿qué ocurre cuando unos elementos devuelven un valor entero y otros devuelven por ejemplo un puntero? Supongamos que un ID devuelve un puntero a la tabla de símbolos, y NUM devuelve un valor float.

Para poder hacer esto, en la sección de declaración se pone:

```
%union {
          Cuerpo              (Al igual que en C)
        }
```

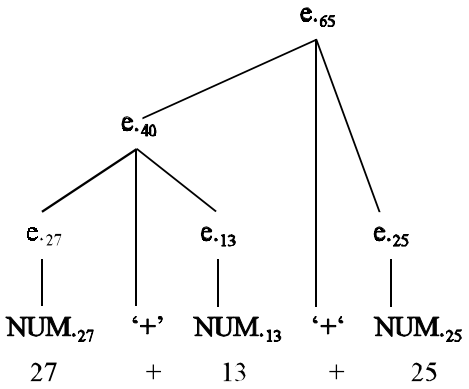
Esto permite definir registros con parte variante. En C se hace con unión, y todo es la parte variante. En el Cuerpo se definen varias variables, y todas comparten el mismo espacio. El espacio ocupado por la unión es el de la variable más grande. En nuestro caso podemos hacer:

```
%union {
          int numero;
          nodo * ptrNodo;
        }
```

ahora nuestro YYSTYPE es o un entero o u n puntero, y los nombres *numero* y *ptrNodo* es como los referenciamos. Además hay que decirle, para *ID* haz uso del campo *ptrNodo* del YYSTYPE, y para *NUM* haz uso del campo *numero* del YYSTYPE.

```
%token <ptrNodo> ID
%token <numero>NUM
```

Analicemos la siguiente entrada: 27 + 13 + 25




Tenemos que decirle de que tipo es el no terminal e, si no lo ponemos nos dará un error diciendo que no tiene identificador asociado

```
%type <numero> e

%%
e : e '+' t
  | t
  ;
t  : t '*' f
  | f
  ;
f  : '(' e ')'
   | ID          {$$ = $1 }   ⇨ No hace falta poner $1.p trNodo porque ya se lo
                              hemos dicho en %token
    ;
```

% token se emplea no sólo para declarar los tokens, sino también para decir cual es el tipo del valor que devuelve.
% type se emplea para indicar el tipo del valor que devuelve un No terminal.

Observar que ahora yylval = atoi (yytext); no es tan fácil porque ahora yylval es una unión. Si quiero cargar el campo número, entonces se hará de la siguiente forma:

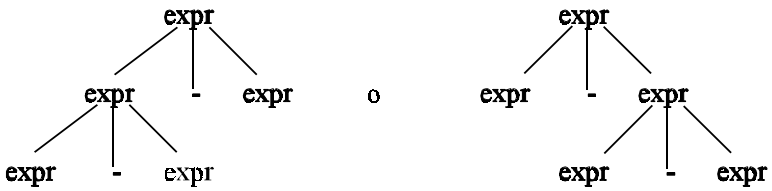
```
%%
[0-9]+  {yylval.numero = atoi(yytext);
         return NUM;}
[A-Z][A-Z0-9]*        {yylval.ptrNodo = .....}
```

Es decir yylval es una variable de tipo registro, y es LEX el encargado de acceder al campo adecuado en cada momento.

✯ **Ambigüedad.**

Yacc no sólo permite expresar de forma fácil una gramática a reconocer, sino que también da herramientas que facilitan la eliminación de ambigüedades.

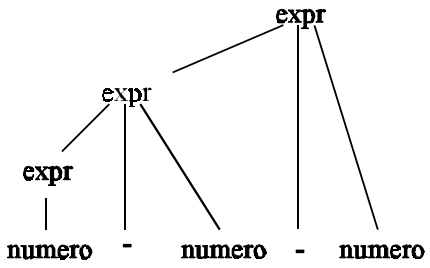
Por ejemplo, supongamos que tenemos una sentencia tal como: *expr : expr - expr*
Se pueden obtener dos árboles sintácticos con la regla *expr : expr '-' expr {$$ = $1 - $3;}*




Si tuviese asociados los valores 30, 15 y 5, en el primer caso daría 10 y en el segundo daría 20.
¿Qué es lo que ocurre?. Como hemos visto en LR, cuando hemos leído *expr - expr*, y nos encontramos el siguiente '-' , podemos o bien reducir *expr - expr* a *expr*, o bien desplazar el siguiente - *expr* y luego empezar a reducir de derecha a izquierda.

Una forma de evitarlo es mediante el cambio de reglas, por ejemplo:

```
expr : expr '-' numero
     | numero
```

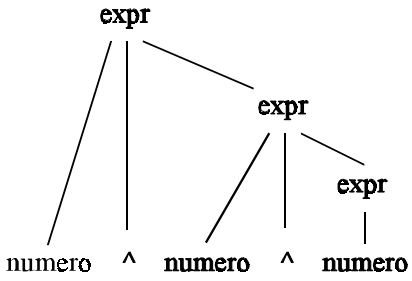



La recursividad por la izquierda implica asociatividad por la izquierda. Si queremos usar asociatividad por la derecha, emplearemos recursividad por la derecha

.

Por ejemplo el operador de esponenciación ^, suele tener asociación a la derecha.

```
expr : numero '^' expr
     | numero
```




Si no cambiamos la gramática, YACC se dará cuenta de este problema cuando compila las reglas, y dará un error reducir/desplazar. YACC nos da una solución alternativa, que soluciona el problema sin necesidad de cambiar la gramática.

A menos que se le ordene lo contrario, YACC resolverá todos los conflictos en las acciones del análisis sintáctico utilizando las dos reglas siguientes:

1. Un conflicto de reducción/reducción se resuelve eligiendo la producción en conflicto que se haya listado primero en la especificación en YACC.

2. Un conflicto de desplazamiento /reducción se resuelve en favor del desplazamiento

La solución más potente es la que YACC nos suministra a través de los operadores de precedencia y asociatividad

%left
%rigth
%nonassoc y %prec

La regla expr ‘-‘ expr dejará de ser ambigua si en la parte de declaraciones se pone
%left ‘-‘

Si ponemos varios tokens en un mismo %left, %right o %nonassoc quiere decir que todos tiene la misma prioridad. Una indicación de la forma:
%left ‘-‘ ’+’
%left ‘*‘ ’/’

indica que + y - tienen la misma prioridad y que son asociativos por la izquierda y que * y / también. Como * y / van después de - y +, esto indica que * y / tienen mayor prioridad que + y -.

%nonassoc sirve para indicar tokens que no se pueden asociar.

Hay casos en los que un operador puede representar varias operaciones, como en el caso del - que puede ser binario o unario. El - solo puede aparecer en un left, right o nonassoc. ¿Cómo solucionamos el problema? En las reglas podemos decir que se aplique una determinada prioridad. Por ejemplo:

```
%left '+' '-'
%left '*' '/'
%%
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr    %prec '*'
```

Un método más directo puede ser declarar un token ficticio con mayor prioridad, por ejemplo:

```
%token MENOS_UNARIO

%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO

%%
expr : expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '-' expr    %prec MENOS_UNARIO
```

✯ **Tratamiento de Errores.**

Quizás una de las mayores deficiencias de YACC es con respecto al tratamiento de errores.

Si se produce un error sintáctico, YACC desecha todos los tokens leídos hasta el momento, pero sólo hasta llegar a un lugar donde el error se pueda recupera. ¿Como podemos recuperar un error? A través del token reservado *error*. Por ejemplo:

```
prog   :      ε
       |      prog sent ';'
       |      prog error ';'
       ;
```

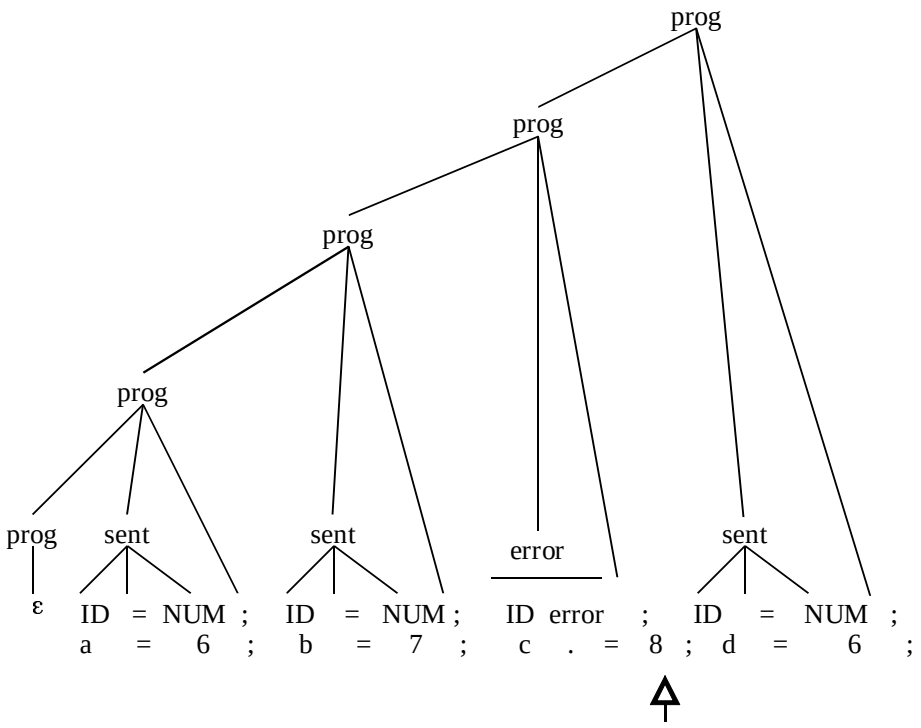
# El generador de analizadores sintáctico: PCYACC

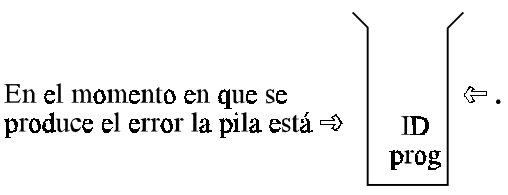
Con la última regla le decimos a yacc que en lugar de alguna sentencia me puedo encontrar un error sintáctico.

Supongamos que tenemos la siguiente sentencia a reconocer:

a = 6; b = 7; c.= 8; d = 6;

```
prog   :    ε
       |    prog sent ';'
       |    prog error ';'
       ;
sent   :    ID '=' NUM
       ;
```



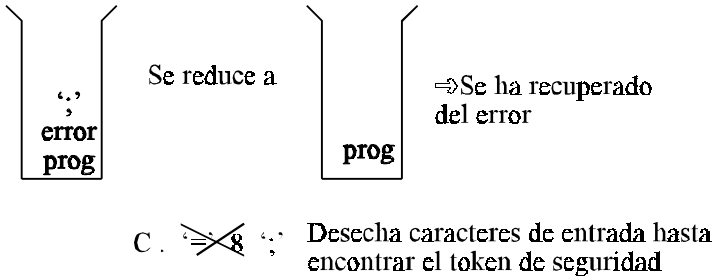
Cuando encuentra el error Yacc, mira lo que hay delante del error.

prog error ';'

Empieza a sacar cosas de la pila hasta encontrar *prog*, después mete *error* en la pila, y empieza a despreciar la entrada hasta encontrar el ';'

| prog error ';'

El ';' es el token de seguridad, éste es muy importante, ya que es el que nos permite recuperarnos del error.




Además para que todo funcione de forma adecuada es conveniente asociarle a la regla de error la invocación de una macro especial de YACC. Esta macro se llama *yyerrok*;

```
prog    :       ε
        |       prog sent ';'
        |       prog error ';'  {yyerrork;}
        ;
sent    :       ID '=' NUM
        ;
```

Con *yyerrork*, le dice a YACC que suponga que se ha llegado ya a una condición segura (condición segura ⇨ correcta recuperación del error). Si no se pone es posible que se pierdan acciones de las otras reglas.

El problema de ese tratamiento de errores es que no se produce una descripción clara del error. Todos los errores que se producen son del mismo tipo: syntax error.

# El generador de analizadores sintáctico: PCYACC

Pasos a seguir para obtener más información cuando se produce un error sintáctico. Información del tipo:

> "Me he encontrado el token (tal) y esperaba un token (cual) : Línea...."

1. En el área de funciones de YACC, hacer *#include "errorlib.c"*

2. Compilar el programa YACC como *pcyacc -d -pyaccpar.c fichero.yac*

   *-d* crea el fichero yytab.h que tiene una descripción de la pila de YACC y el número asociado a cada token.

   *-pyaccpar.c* Hace que se utilice un esqueleto para contener al autómata finito con pila que implementa YACC. En lugar de tomar el esqueleto por defecto, se toma a yaccpar.c que contiene las nuevas funciones de tratamiento de errores.

   *yaccpar.c* hace un include de errorlib.h

3. Ejecutar el programa *tokens*.
   *tokens* toma el fichero *yytab.h*, y genera *yytok.h* que no es más que la secuencia de nombres de tokens entrecomillada, y separada por comas.

   *yytok.h* es utilizado por errorlib.c para indicar qué tokens se esperaban.

4. Compilar *fichero.c* como siempre
   *fichero.exe* dice en la línea que se ha producido el error y la descripción del error.

**El área de funciones**. La tercera parte de una especificación en YACC consta de rutinas de apoyo escritas en C. Se debe proporcionar un análisis léxico llamado yylex(). En caso necesario se pueden agregar otros procedimientos, como rutinas de recuperación de errores.

El analizador léxico yylex () produce pares formados por un token y su valor de atributo asociado. Si se devuelve un token como DIGITO, dicho token se debe declarar en la primera sección de la especificación en YACC, El valor del atributo asociado a un token se comunica al analizador sintáctico mediante una variable yylval.

En concreto, debemos especificar dos funciones mínimo:

- main( ), que deberá arrancar el analizador haciendo una llamada a yyparse( ).
- yyerror(s) char *s; que recibe una cadena que describe el error. En este momento yychar contiene el terminal que se ha recibido y que no se esperaba. (El yyerror(s), sólo es necesario si no se han hecho los cuatro pasos anteriores).