Capítulo 4

Gramáticas atribuidas

4.1 Análisis semántico

Además de controlar que un programa cumple con las reglas de la gramática del lenguaje, 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 que intervienen en las expresiones sean compatibles o que los parámetros reales de una función sean coherentes con los parámetros formales: p.ej. no suele tener mucho sentido el multiplicar una cadena de caracteres por un entero.

Comenzaremos viendo un ejemplo sencillo en el que se introduce el concepto de atributo mediante la construcción del intérprete de una calculadora.

4.1.1 Atributos y acciones semánticas

A continuación se desarrolla un ejemplo para tener una visión global de este tema. Supongamos que se desea construir una pequeña calculadora que realiza las operaciones + y * . La gramática será la siguiente:

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

y queremos calcular: 33 + 12 + 20. Los pasos a seguir son:

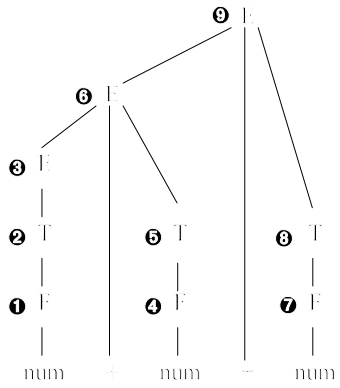


Figure 1Árbol de reconocimiento de "num + num + num". Los números entre círculos indican la secuencia de reducciones según un análisis LR(1)

1. Análisis léxico. La cadena de entrada se convierte en una secuencia de *tokens*, p.ej. con PCLex: 33 + 12 + 20 ➪ num + num + num

2. Análisis sintáctico. Mediante la gramática anterior y, p. ej., PCYacc se produce el árbol sintáctico de la figura 4.1.

Hemos conseguido construir el árbol sintáctico, lo que no quiere decir que sea semánticamente correcto. Si deseamos construir un intérprete, el siguiente paso es saber qué resultado nos da cada operación, y para ello hemos etiquetado el árbol indicando el orden en el que se han aplicado las reducciones. El orden en el que se van aplicando las reglas nos da el *parse* derecho.

El orden en que se aplican las reglas de producción está muy cercano a la semántica de lo que se va a reconocer. Y este hecho debemos aprovecharlo. Para ello vamos a decorar el árbol sintáctico asociando valores a los terminales y no terminales que en él aparecen. A los terminales le asignaremos ese valor mediante una acción asociada al patrón correspondiente en el análisis léxico, de la siguiente forma:

```
[0-9]+    {
          Convertir yytext en un entero y asociar dicho entero al token NUM;
          return NUM;
          }
```

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

$$num._{33} +_{\cdot nada}\ num._{12} +_{\cdot nada}\ num._{20}$$

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

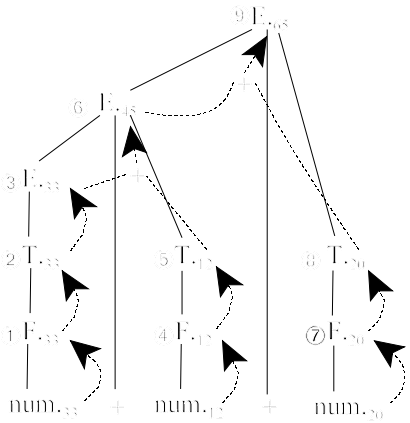


Figure 2 Propagación de atributos de las hojas hacia la raíz

idea es que cada no terminal de la gramática tenga asociado un atributo que aglutine, de alguna manera, la información del sub-árbol sintáctico del que es raiz.

La figura 4.2 ilustra la propagación de atributos desde los terminales hacia los no terminales. Esta propagación puede implementarse ejecutando acciones asociadas a cada regla de producción, de tal manera que la acción se ejecutará cada vez que se reduzca por su regla asociada, de igual manera que podíamos asociar una acción a un patrón del analizador léxico. De esta manera, con las asociaciones de valores a los terminales y no terminales y las asociaciones de **acciones semánticas** a los patrones y a las reglas de producción, podemos evaluar el comportamiento de un programa, es decir, generamos código (concreto o abstracto; podemos considerar que un intérprete genera código abstracto, ya que en ningún momento llega a existir físicamente). Esto puede verse en la siguiente tabla (los subíndices tan sólo sirven para diferenciar entre distintas instancias de un mismo símbolo gramatical cuando aparece varias veces en la misma regla).

Reglas de producción	Acciones semánticas asociadas
$E_1 \rightarrow E_2 + T$	$\{E_1 = E_2 + T;\}$
$E \rightarrow T$	$\{E = T;\}$
$T_1 \rightarrow T_2 * F$	$\{T_1 = T_2 * F;\}$
$T \rightarrow F$	$\{T = F;\}$
$F \rightarrow (E)$	$\{F = E;\}$
$F \rightarrow num$	$\{F = NUM;\}$

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

Basándonos en el funcionamiento de un analizador con funciones recursivas, es como si cada función retornase un valor que representa al sub-árbol del que es raíz. La implementación en el caso LR(1) es un poco más compleja, almacenándose los atributos en la pila α de manera que, excepto por el estado inicial, α almacena tuplas de la forma (estado, símbolo, atributo), con lo que todo símbolo tiene un atributo asociado incluso cuando éste no es necesario para nada (como en el caso del *token* + del ejemplo anterior). Los cambios producidos se ilustran en la figura 4.3.

Así, un **atributo** es una información asociada a un terminal o a un no terminal. Una **acción** o **regla semántica** es un algoritmo que puede acceder a los atributos de los terminales y/o no terminales. Como acción semántica no sólo se puede poner una asignación a un atributo, además puede añadirse código; p.ej. si se hace la asociación:

① $E_1 \rightarrow E_2 + T$ $\{E_1 = E_2 + T;$ printf("%d\n", $E_1)\}$

utilizando notación pseudo-C, entonces cada vez que se aplique la regla ① se visualizará en pantalla el valor del atributo de la regla que, en el ejemplo, serán los valores 45 y 65. Si sólo se deseara que apareciera el valor resultante global, podría añadirse una nueva regla S → E y pasar el **printf** de la regla ① a esta otra:

S → E { printf("%d\n", E); }

$E_1 \rightarrow E_2 + T \qquad \{E_1 = E_2 + T; \}$

De esta forma, la construcción del árbol sintáctico tendría una reducción más, y cuando ésta se hiciera aparecería el resultado por pantalla. Nótese como la adición de esta regla no modifica la gramática en el sentido de que sigue reconociendo el mismo lenguaje pero, sin embargo, nos proporciona una clara utilidad al permitir visualizar únicamente el resultado final del cálculo.

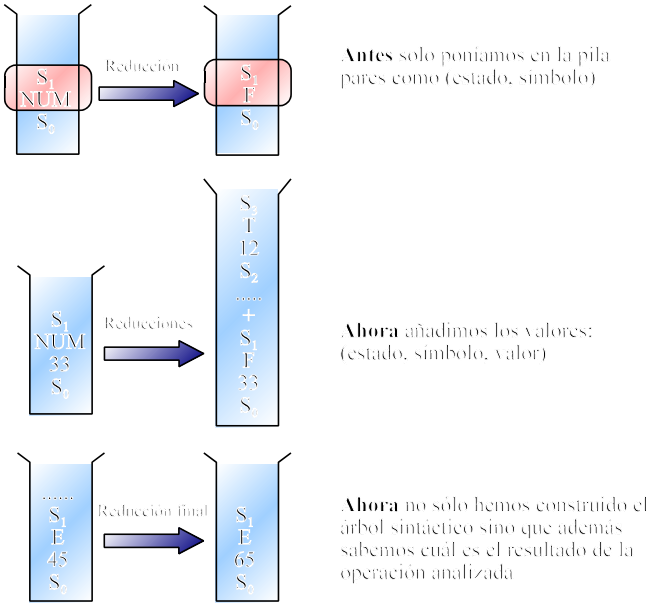


Figure 3 Gestión de la pila α sin atributos y con atributos

4.1.2 Ejecución de una acción semántica

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

- **Definición dirigida por sintaxis**: consiste en asociar una acción semántica a una regla de producción, pero dicha asociación no indica cuándo se debe ejecutar dicha acción semántica. Se supone que en una primera fase se construye el árbol sintáctico completo y, posteriormente, se ejecutan las acciones semánticas en una secuencia tal que permita el cálculo de todos los atributos de los nodos del árbol.
- **Esquema de traducción**: es igual que una definición dirigida por sintaxis excepto que, además, se asume o se suministra información acerca de cuándo se deben ejecutar las acciones semánticas. Es más, también es posible intercalar acciones entre los símbolos del consecuente de una regla de

producción, ej.: A → c d {A = c + d} B. En el caso LR las acciones se ejecutan cuando se efectúan las reducciones. En el caso con funciones recursivas las acciones se encuentran intercaladas con el código destinado a reconocer cada expresión BNF.

Siempre que tengamos un esquema de traducción con acciones intercaladas se puede pasar a su equivalente definición dirigida por sintaxis. Por ejemplo, el esquema:

$A \rightarrow c\ d\ \{ \text{acción}_1 \}\ B$

se puede convertir en

$A \rightarrow c\ d\ N_1\ B$

$N_1 \rightarrow \varepsilon\ \{ \text{acción}_1 \}$

que es equivalente. Básicamente, la transformación consiste en introducir un no terminal ficticio (N_1 en el ejemplo) que sustituye a cada acción semántica intermedia (acción_1), y crear una nueva regla de producción de tipo épsilon con dicho no terminal como antecedente y con dicha acción semántica como acción.

A primera vista esta transformación puede parecer que puede dar problemas; p.ej. al convertir el esquema:

$A \rightarrow c\ d\ \{$ printf("%d\n", c+d); $\}\ B$

en:

$A \rightarrow c\ d\ N_1\ B$

$N_1 \rightarrow \varepsilon\ \{$ printf("%d\n", c+d); $\}$

aparece una acción semántica que hace referencia a los atributos de c y d asociada a una regla en la que c y d no intervienen. Sin embargo, dado que sólo existe una regla en la que N_1 aparece en el antecedente y también sólo una regla en la que N_1 aparece en el consecuente, está claro que siempre que un analizador intente reducir a N_1 o invocar a la función recursiva que implementa a N_1, los dos últimos *tokens* que hayan sido consumidos serán c y d, por lo que la acción semántica puede ejecutarse trabajando sobre los atributos de estos dos últimos *tokens*, aunque aparentemente parezca que hay algún problema cuando se echa un vistazo a la ligera a la definición dirigida por sintaxis obtenida tras la transformación del esquema de traducción.

Cuando se trabaja con reglas de producción (no con funciones recursivas), podemos encontrarnos con algún otro problema. P.ej., si se tiene:

$A \rightarrow c\ d\ \{A = c + d\}\ B$

y se transforma a:

$A \rightarrow c\ d\ N_1\ B$

$N_1 \rightarrow \varepsilon\ \{A = c + d\}$

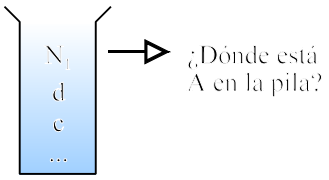


Figure 4 Esquema de la pila α tras reducir por la regla $N_1 \rightarrow \varepsilon$. La acción semántica no puede acceder al atributo de A porque A no ha sido reducida todavía y no está en la pila

entonces en la pila se tiene lo que ilustra la figura 4.4.

En general, la utilización de atributos en un esquema de traducción obedece a reglas del sentido común una vez que se conocen las transformaciones que un metacompilador aplica automáticamente a las acciones semántica intermedias, ya se trate de analizadores LR o con funciones recursivas. En el caso del análisis ascendente la cosa puede no estar tan clara de primera hora y resulta conveniente exponer claramente cuáles son estas **reglas sobre la utilización de atributos en acciones semánticas de un esquema LR**:

- Un atributo solo puede ser usado (en una acción) detrás del símbolo al que pertenece.
- El atributo del antecedente solo se puede utilizar en la acción (del final) de su regla. Ej.:

  ✔ $A \rightarrow c\ d\ N_1\ B$ { sólo aquí se puede usar el atributo de A }

  ✘ $N1 \rightarrow \varepsilon$ { aquí no se puede usar el atributo de A }
- En una acción intermedia sólo se puede hacer uso de los atributos de los simbolos que la preceden. Siguiendo los ejemplo anteriores, si se hace una sustitución de una acción intermedia y se obtiene:

  $A \rightarrow c\ d\ N_1$

  $N_1 \rightarrow \varepsilon$

  pues en la acción de N_1 sólo se puede hacer uso de los atributos de c y d (que es lo que le precede).

  Todo esto se verá en epígrafes posteriores con mayor nivel de detalle.

4.2 Traducción dirigida por sintaxis

A partir de este epígrafe se desarrolla la traducción de lenguajes guiada por gramáticas de contexto libre. Se asocia información a cada construcción del lenguaje de programación proporcionando atributos a los símbolos de la gramática que representan dicha construcción. Los valores de los atributos se calculan mediante reglas (acciones) semánticas asociadas a las reglas de producción gramatical.

Como hemos visto, hay dos notaciones para asociar reglas semánticas con reglas de producción:

- 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 ejecución de las acciones.
- 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, como p.ej. evaluar expresiones aritméticas en una calculadora. La traducción de la cadena de componentes léxicos es el resultado obtenido al evaluar las reglas semánticas.

En la práctica, no todas las implementaciones tienen que seguir al pie de la letra este esquema tan nítido de una fase detrás de otra. 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.

4.2.1 Definición dirigida 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 que cada nodo de un árbol sintáctico tiene asociado un registro con campos para guardar información, entonces un atributo corresponde al nombre de un campo.

Un atributo puede almacenar cualquier cosa: una cadena, un número, un tipo, una posición de memoria, etc. El valor de un atributo en un nodo de un árbol sintáctico se define mediante una regla semántica asociada a la regla de producción utilizada para obtener dicho nodo. El valor de un **atributo sintetizado** de un nodo se calcula a partir de los valores de los atributos de sus nodos hijos en el árbol sintáctico; el valor de un **atributo heredado** se calcula a partir de los valores de los atributos en los nodos hermanos y padre. Cada nodo del árbol representa una instancia de un símbolo gramatical y, por tanto, tiene sus propios valores para cada atributo.

Las **reglas semánticas** establecen las dependencias entre los atributos; dependencias que pueden representarse mediante un grafo, una vez construido el árbol sintáctico al completo. Del grafo de dependencias se obtiene un orden de evaluación de todas 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, como por ejemplo visualizar un valor o actualizar una variable global. Como se indicó anteriormente, en la práctica, un traductor 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 sintáctico que muestre los valores de los atributos en cada nodo se denomina un **árbol sintáctico decorado** o con anotaciones. El proceso de calcular los valores de los atributos en los nodos se denomina anotar o **decorar el árbol sintáctico**.

4.2.2 Esquema formal de una definición dirigida por sintaxis

Formalmente, en una definición dirigida por sintaxis, cada regla de producción

$A \rightarrow \alpha$

tiene asociado un conjunto de reglas semánticas siendo cada una de ellas de la forma

$b := \mathbf{f}(c_1, c_2, ..., c_k)$

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 α, o bien

2.- *b* es un **atributo heredado** de uno de los símbolos de α , y $c_1, c_2, ..., c_k$ son atributos de los restantes símbolos de α 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 asignaciones de las reglas semánticas a menudo se escriben 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 antecedente de la regla de producción asociada, de manera que no se muestran ni el atributo ficticio ni el signo := de la regla semántica.

P.ej., la definición dirigida por sintaxis del cuadro 4.1 es para un traductor que implementa una calculadora. Esta definición asocia un atributo sintetizado val con valor entero 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 antecedente a partir de los valores de los atributos del consecuente.

El componente léxico num tiene un atributo sintetizado val_lex cuyo valor viene proporcionado por el analizador léxico. La regla asociada a la producción

$L \rightarrow E$ '\n'

donde L es el axioma inicial, es sólo un procedimiento que visualiza 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 PCYacc para esta calculadora, y ello nos llevará a ilustrar la traducción durante el

Reglas de producción	Acciones semánticas asociadas
$L \rightarrow E$ '\n'	printf("%d\n", E.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
$F \rightarrow$ num	F.val = num.val_lex

Cuadro 4.1 Definición dirigida por sintaxis de una calculadora sencilla

análisis sintáctico LALR(1). También veremos un ejemplo con JavaCC pero, por ahora, seguiremos profundizando en las definiciones dirigidas por sintaxis.

En una definición dirigida por sintaxis, se asume que los *tokens* sólo tienen atributos sintetizados, ya que la definición no proporciona ninguna regla semántica para ellos. 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, ya que en muchos casos suele carecer de padre y de hermanos.

4.2.2.1 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**. Con una definición con atributos sintetizados se puede decorar un árbol sintáctico mediante la evaluación de las reglas semánticas de forma ascendente, calculando los atributos desde los nodos hoja a la raíz.

Por ejemplo, la definición con atributos sintetizados del cuadro 4.1 especifica una calculadora que lee una línea de entrada que contiene una expresión aritmética que incluye dígitos (asumimos que el *token* num está asociado al patrón [0-9]+), paréntesis, los operadores + y *, seguida de un carácter de retorno de carro('\n'), e imprime el valor de la expresión. Por ejemplo, dada la cadena "3*5+4" seguida de un retorno de carro, el sistema visualiza el valor 19. La figura 4.5 contiene el árbol sintáctico decorado para la entrada propuesta. El resultado, que se imprime en la raíz del árbol, es el valor de E.val.

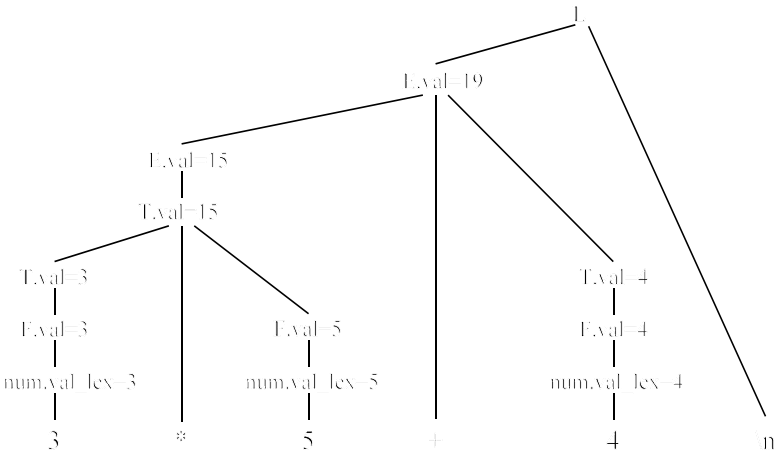


Figure 5Árbol de análisis sintáctico con anotaciones para reconocer "3*5+4\n".

Aunque una definición dirigida por sintaxis no dice nada acerca de cómo ir

calculando los valores de los atributos, está claro que debe existir una secuencia viable de cálculos que permita realizar dichos cálculos. Vamos a ver una posible secuencia. Considérese el nodo situado en el extremo inferior izquierdo que corresponde al uso de la producción F → num. La regla semántica correspondiente, F.val:= num.val_lex, establece que el atributo F.val en el nodo tenga el valor 3 puesto que el valor de num.val_lex en el hijo de este nodo es 3. De forma similar, en el padre de este F, el atributo T.val tiene el valor 3.

A continuación considérese el nodo raíz de la producción T → T * F. El valor del atributo T.val en este nodo está definido por T.val := T_1.val * F.val. Cuando se aplica la regla semántica en este nodo, T_1.val tiene el valor 3 y F.val el valor 5. Por tanto, T.val adquiere el valor 15 en este nodo.

Por último, la regla asociada con la producción para el no terminal inicial, L → E \n, visualiza el valor de la expresión representada por E, esto es E.val.

4.2.2.2 Atributos heredados

Un atributo heredado es aquél cuyo valor en un nodo de un árbol de análisis sintáctico está definido a partir de los atributos de su padre y/o de sus hermanos. 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. El cuadro 4.2 muestra una gramática que permite una declaración, generada por el axioma D, formada por la palabra clave int o real seguida de una lista de identificadores separados por comas.

Reglas de producción	Acciones semánticas asociadas
D → T L	L.her:= T.tipo
T → int	T.tipo := “integer”
T → real	T.tipo := “real”
L_2 → L_1 , id	L_1.her := L_2.her añadetipo(id.ptr_tds, L_2.her)
L → id	añadetipo(id.ptr_tds, L.her)

Cuadro 4.2 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 regla por la que se genera. 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 hacia abajo este valor por el árbol sintáctico utilizando el atributo heredado L.her. Las reglas asociadas con las producciones de L llaman al procedimiento añadetipo, que suponemos que añade el tipo de cada identificador a su entrada en la tabla de símbolos (apuntada por el atributo ptr_tds). Este último detalle carece de importancia para el asunto que estamos tratando en este momento y se verá con mayor detenimiento en capítulos posteriores.

En la figura 4.6 se muestra un árbol de análisis sintáctico con anotaciones para la cadena "real id_1, id_2, id_3". El valor de L.her en los tres nodos de L proporciona el tipo a los identificadores id_1, id_2 e id_3. Estos valores se obtienen 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 figura. En cada nodo de L también se llama al procedimiento añadetipo para indicar en la tabla de símbolos el hecho de que el identificador en el hijo de cada nodo L tiene tipo real.

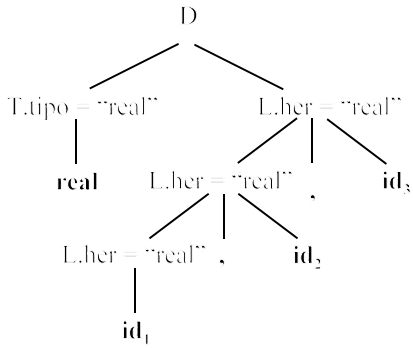


Figure 6Árbol sintáctico con el atributo heredado L.her en cada nodo L

4.2.2.3 Grafo de dependencias

Si un atributo *b* en un nodo de un árbol 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 en los nodos de un árbol sintáctico pueden representarse mediante un grafo dirigido llamado **grafo de dependencias**.

Antes de construir el grafo de dependencias para un árbol 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 a procedimiento. El grafo tiene un nodo por cada atributo de un nodo del árbol y una arista desde el nodo *c* al *b* si el atributo *b* depende del atributo *c*. Algorítmicamente, el grafo de dependencias para un determinado árbol de análisis sintáctico se construye de la siguiente manera:

Para cada nodo *n* en el árbol de análisis sintáctico

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

Por ejemplo, supóngase que A.a := f(X.x, Y.y) es una regla semántica para la producción A → X Y. Esta regla define un atributo sintetizado *A.a* que depende de los atributos *X.x* e *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 un arco desde *X.x* hasta *A.a* puesto que *A.a* depende de *X.x* y otro arco desde *Y.y* hasta *A.a* puesto que *A.a* también depende de *Y.y*.

Si la regla de producción A→ XY tuviera asociada la regla semántica X.i := g(A.a, Y.y), entonces habría un arco desde *A.a* hasta *X.i* y otro arco desde *Y.y* hasta *X.i*, puesto que *X.i* depende tanto de *A.a* como de *Y.y*.

Los arcos que se añaden a un grafo de dependencias son siempre los mismos para cada regla semántica, o sea, siempre que se utilice la regla de producción del

Regla de producción	Acción semántica asociada
$E_3 \rightarrow E_1 + E_2$	$E_3.val := E_1.val + E_2.val$

Cuadro 4.3 Ejemplo de regla de producción con acción semántica asociada

cuadro 4.3 en un árbol sintáctico, se añadirán al grafo de dependencias los arcos que se muestran en la figura 4.7.

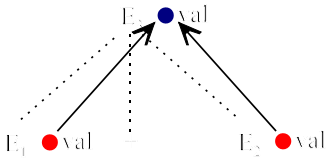


Figure 7 $E_3.val$ se sintetiza a partir de $E_1.val$ y $E_2.val$

Los tres nodos del grafo de dependencias (marcados con un círculo de color) representan los atributos sintetizados $E_3.val$, $E_1.val$ y $E_2.val$ en los nodos correspondientes del árbol sintáctico. El arco desde $E_1.val$ hacia $E_3.val$ muestra que $E_3.val$ depende de $E_1.val$ y el arco hacia $E_3.val$ desde $E_2.val$ muestra que $E_3.val$ también depende de $E_2.val$. Las líneas de puntos representan al árbol de análisis sintáctico y no son parte del grafo de dependencias.

En la figura 4.8 se muestra el grafo de dependencias para el árbol de análisis sintáctico de la figura 4.6. Los arcos de este grafo están numerados. Hay una arco(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 de la regla de producción D → T L. Los dos arcos que apuntan hacia abajo (④ y ⑥) surgen porque L_{i+1}.her depende de L_i.her según la regla semántica L_1.her := L_2.her de la regla de producción $L_2 \rightarrow L_1$, 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 amarillos se construyen para dichos falsos atributos (arcos ③, ⑤ y ⑦).

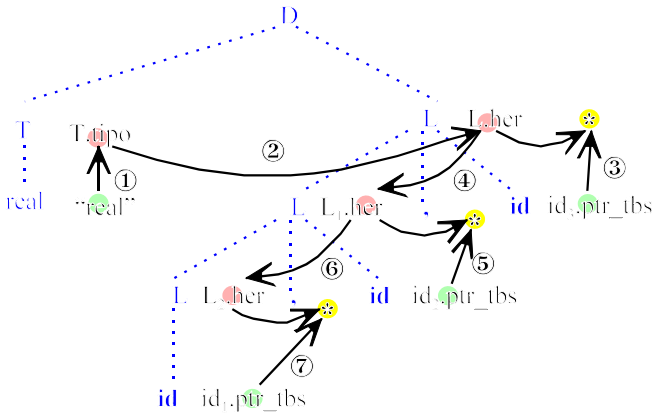


Figure 8 Grafo de dependencias para el árbol sintáctico de la figura 4.6. El árbol sintáctico aparece de azul, los atributos de los *tokens* de verde, los atributos ficticios de amarillo, y el resto de atributos se muestra en rosa.

4.2.2.4 Orden de evaluación

Una ordenación topológica de un grafo dirigido acíclico es toda secuencia m_1, m_2, ..., m_k de los nodos del grafo tal que los arcos van desde los nodos que aparecen primero en la ordenación a los que aparecen más tarde; es decir, si $m_i \rightarrow m_j$ es un arco desde m_i a m_j, entonces m_i aparece antes que m_j en la ordenación.

Toda ordenación topológica 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 sintáctico. Es decir, en el ordenamiento topológico, los atributos dependientes c_1, c_2,..., c_k en una regla semántica 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:

1.- Se utiliza la gramática subyacente para construir un árbol sintáctico para la entrada.

2.- El grafo de dependencias se construye como se indica más arriba.

3.- A partir de una ordenación topológica 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 y/o desencadena su interpretación.

Por ejemplo, la numeración de arcos que se hizo en el grafo de dependencias de la figura 4.8, nos suministra una ordenación topológica lo que produce la secuencia:

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 finalmente el tipo "real" en la entrada de la tabla de símbolos para cada identificador.

Por último, se dice que una gramática atribuida es **circular** si el grafo de dependencias para algún árbol sintáctico generado por la gramática tiene un ciclo. Por ejemplo, si una regla como:

A→ B C D

tiene las reglas semánticas:

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

entonces cualquier aplicación de la regla dará lugar a un trozo de árbol sintáctico y grafo de dependencias como el de la figura 4.9.

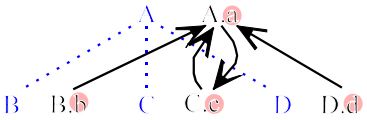


Figure 9 Grafo de dependencias circular

Resulta evidente que si el grafo de dependencias tiene un ciclo entonces no es posible encontrar un orden de evaluación. En cualquier otra situación existe, al menos, una ordenación topológica.

4.2.2.5 Gramática L-atribuida

También llamada gramática atribuida por la izquierda, o definición con atributos por la izquierda, es aquélla en la que en toda regla A → X_1 X_2 ... X_n, cada atributo heredado de X_j, con $1 \leq j \leq n$, 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 produce ciclos y además admiten un orden de evaluación en profundidad mediante el siguiente algoritmo:

Función visitaEnProfundidad(nodo n){
Para cada hijo m de n, de izquierda a derecha

evaluar los atributos heredados de m
visitaEnProfundidad(m)
Fin Para
evaluar los atributos sintetizados de n
Fin visitaEnProfundidad

A continuación se muestra una gramática L-atribuida donde D hace referencia a "declaración", T a "tipo" y L a "lista de identificadores".

D → T L	{L.tipo = T.tipo;}
T → integer	{T.tipo = "integer";}
T → real	{T.tipo = "real";}
L → L_1 , id	{L_1.tipo = L.tipo; id.tipo = L.tipo;}
L → id	{id.tipo = L.tipo;}

Sin embargo, la siguiente gramática no es L-atribuida ya que propaga los atributos de derecha a izquierda:

D → id T	{id.tipo = T.tipo; D.tipo = T.tipo;}
D → id D_1	{id.tipo = D_1.tipo; D.tipo = D_1.tipo;}
T → integer	{T.tipo = "integer";}
T → real	{T.tipo = "real";}

4.2.2.6 Gramática S-atribuida

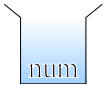
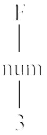
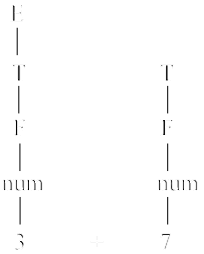
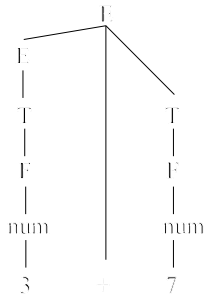
Es una gramática atribuida que sólo contiene atributos sintetizados.Una gramática S-atribuida es también L-atribuida. Los atributos sintetizados se pueden evaluar con un analizador sintáctico ascendente conforme se va construyendo el árbol sintáctico. El analizador sintáctico puede conservar en su pila 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 regla de producción con la que se reduce.

El cuadro 4.4 muestra un ejemplo de gramática S-atribuida, y la tabla siguiente

Regla de producción	Acción semántica asociada
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 → num	F.val = num.val lex

Cuadro 4.4 Ejemplo de gramática S-atribuida

un ejemplo con el que se pone de manifiesto cómo se propagan los atributos, suponiendo la entrada "3 + 7 \n".

Árbol	Pila α	Acción y atributos
		num.val_lex = 3
		F.val = num.val_lex (F.val = 3)
		T.val = 7 +.nada = -- E.val = 3
		E.val = E_1.val + T.val (E.val = 3 + 7 = 10)

Como ya se ha adelantado, la pila además de tener estados y símbolos, también almacena intercaladamente los atributos.

4.2.3 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 la regla

de producción. Por ejemplo:

$$A \rightarrow cd \ \{printf(c + d)\} \ B$$

Cuando se diseña un esquema de traducción, se deben respetar algunas limitaciones para asegurarse que el valor de un atributo esté disponible cuando una acción se refiera a él. Estas limitaciones, derivadas del tipo de análisis sintáctico escogido para construir el árbol, 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 regla de producción $T \rightarrow T_1 * F$ y la acción semántica $T.val := T_1.val * F.val$ 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, de manera que deben cumplirse algunas reglas a la hora de utilizar los atributos de los símbolos gramaticales:

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 debe colocar, generalmente, al final del lado derecho de la producción.

Los esquemas de traducción bien definidos que cumplen estas restricciones se dice que están **bien definidos**. Ejemplo:

$$D \rightarrow T \ \{L.her = T.tipo\} \ L \ ;$$
$$T \rightarrow int \ \{T.tipo = \text{“integer”}\}$$
$$T \rightarrow real \ \{ T.tipo = \text{“real”}\}$$
$$L \rightarrow \{L_1.her = L.her\} \ L_1 \ , \ id \ \{añadetipo \ (id.ptr\_tbs, L.her)\}$$
$$L \rightarrow id \quad \{añadetipo(id.ptr\_tbs, L.her)\}$$

4.2.4 Análisis LALR con atributos

En este epígrafe se estudiará cómo funciona el análisis LALR(1) con atributos. Recordemos que el LALR(1) es capaz de saber en todo momento qué regla aplicar sin margen de error (ante una gramática que cumpla las restricciones LALR(1)). Este hecho es el que se aprovecha para pasar a ejecutar las acciones de forma segura. En el caso de retroceso, sería 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.

4.2.4.1 Conflictos reducir/reducir

Hay esquemas de traducción que no pueden ser reconocidos por analizadores LALR(1) directamente. En concreto, se suelen ejecutar acciones semánticas únicamente cuando se hace una reducción, y nunca cuando se desplaza. Por ello, un esquema de traducción con acciones intercaladas se convierte en otro equivalente que sólo contiene acciones al final de la regla, tal y como se ilustró en el epígrafe 4.1.2. A pesar de ello, esta transformación puede conducir a situaciones que dejan de ser LALR(1), como se ilustra en la figura 4.10.

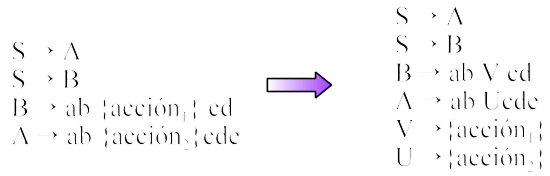


Figure 10 Ejemplo de gramática que no es LALR(1) después de haber sido transformada con reglas épsilon.

En este caso se produce un conflicto reducir/reducir, porque la gramática no es LALR.Como ya se ha indicado un conflicto reducir/reducir se produce cuando llega un momento del análisis en el que no se sabe que regla aplicar. En el caso anterior, si se introduce la cadena "abcd", cuando se hayan consumido los dos primeros *tokens* ("a" y "b") no podemos saber que regla ε aplicar, si U o V, ya que el *token* de pre-búsqueda es "c" y éste sucede tanto a U como a 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 EOF para saber por qué regla reducir y, por tanto, qué acción aplicar.

4.2.4.2 Conflictos desplazar/reducir

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

S → aBadd
S → aCd
B → a
C → aa

Al leer la cadena "aaad", cuando se consumen las dos primeras "a" y el *token* de prebúsqueda es la tercera "a", no se sabe si reducir la última "a" consumida a B o

desplazar para reducir a continuación a C. Todo depende de con cuántas "d" finalice la cadena lo cual aún se desconoce.

Un ejemplo de gramática a la que le sucede algo parecido es la correspondiente al reconocimiento de sentencias if anidadas en el lenguaje C. La gramática es:

S → if COND then S
S → if COND then S else S
S → id = EXPR

en la que se producen conflictos desplazar/reducir ante una entrada como:

if $COND_1$ **then if** $COND_2$ **then** S_1 **else** S_2

tal y como ilustra la figura 4.11.

Cuando un compilador de C se encuentra una sentencia como ésta opta por construir el segundo árbol sintáctico siguiendo la regla general "emparejar cada else con el then sin emparejar anterior más cercano". Esta regla para eliminar ambigüedades se puede incorporar directamente a la gramática añadiendo algunas reglas de producción adicionales:

S → SCompleta
| SIncompleta

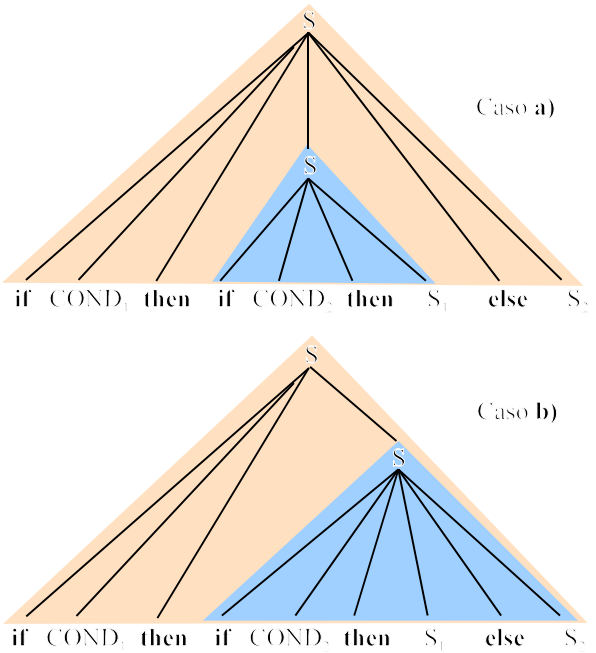


Figure 11 Ambigüedad en dos if anidados en C:

caso **a)** else asociado al if externo

caso **b)** else asociado al if interno

SCompleta → if COND then SCompleta else SCompleta
| id = EXPR
SIncompleta → if COND then S
| if COND then SCompleta else SIncompleta

que asocian siempre el else al if más inmediato. De forma que una SCompleta es o una proposición if-then-else que no contenga proposiciones incompletas o cualquier otra clase de proposición no condicional (como por ejemplo una asignación). Es decir, dentro de un if con else solo se permiten if completos, lo que hace que todos los else se agrupen antes de empezar a reconocer los if sin else.

4.3 El generador de analizadores sintácticos PCYacc

PCYacc es un metacompilador que acepta como entrada una gramática de contexto libre, y produce como salida el autómata finito que reconoce el lenguaje generado por dicha gramática. Aunque PCYacc no acepta todas las gramáticas sino sólo las LALR (*LookAhead LR*), la mayoría de los lenguajes de programación se

pueden expresar mediante una gramática de este tipo. Además permite asociar acciones a cada regla de producción y asignar un atributo a cada símbolo de la gramática (terminal o no terminal). Esto facilita al desarrollador la creación de esquemas de traducción en un reconocimiento dirigido por sintaxis. Dado que el autómata que genera PCYacc está escrito en lenguaje C, todas las acciones semánticas y declaraciones necesarias deben ser escritas por el desarrollador en este mismo lenguaje.

PCYacc parte de unos símbolos terminales (*tokens*), que deben ser generados por un analizador lexicográfico que puede implementarse a través de PCLex, o bien *ad hoc* siguiendo algunas convenciones impuestas por PCYacc.

4.3.1 Formato de un programa Yacc

Al lenguaje en el que deben escribirse los programas de entrada a PCYacc lo denominaremos Yacc. Así, 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 que permite reconocer expresiones aritméticas:

E → E + T | T
T → T * F | F
F → M (E) | M id | M num
M → - | ε

A continuación se describirán progresivamente cada una de estas áreas.

4.3.1.1 Área de definiciones

Este área está compuesta por dos secciones opcionales. La primera de ellas está delimitada por los símbolos **%{** y **%}** y permite codificar declaraciones ordinarias en C. Todo lo que aquí se declare será visible en las áreas de reglas y de funciones, por lo que suele usarse para crear la tabla de símbolos, o para incluirla (**#include**). Los símbolos **%{** y **%}** deben comenzar obligatoriamente en la columna 0 (al igual que todas las cláusulas que se verán a continuación)

La segunda sección permite declarar los componentes léxicos (*tokens*) usados en la gramática. Para ello se utiliza la cláusula **%token** seguida de una lista de terminales (sin separarlos por comas); siguiendo el ejemplo del punto 4.3.1 habría que declarar:

```
%token ID NUM
```

Además, a efectos de aumentar la legibilidad, pueden utilizarse tantas cláusulas **%token** como sean necesarias; por convención, los *tokens* se escriben en mayúsculas.

Internamente, PCYacc codifica cada *token* como un número entero,

empezando desde el 257 (ya que del 0 al 255 se utilizan para representar los caracteres ASCII, y el 256 representa al *token* de error). Por lo tanto la cláusula

%token ID NUM

es equivalente a

define ID **nº**$_1$
define NUM **nº**$_2$

Evidentemente, los números o códigos asignados por PCYacc a cada *token* son siempre distintos.

No obstante lo anterior, no todos los terminales han de enumerarse en cláusulas **%token**. En concreto, los terminales se pueden expresar de dos formas:

- Si su lexema asociado está formado por un solo carácter, y el analizador léxico lo devuelve tal cual (su código ASCII), entonces se pueden indicar directamente en una regla de producción sin declararlo en una cláusula **%token**. Ejemplo: '+'.
- Cualquier otro terminal debe declararse en la parte de declaraciones, de forma que las reglas harán referencia a ellos en base a su nombre (no al código, que tan sólo es una representación interna).

Finalmente, en este área también puede declararse el axioma inicial de la gramática de la forma:

%start noTerminal

Si esta cláusula se omite, PCYacc considera como axioma inicial el antecedente de la primera regla gramatical que aparece en el área de reglas.

4.3.1.2 Creación del analizador léxico

Los analizadores léxicos los crearemos a través de la herramienta PCLex, y no *ad hoc*, con objeto de facilitar el trabajo. En el caso del ejemplo propuesto en el punto 4.3.1, se deberán reconocer los identificadores y los números, mientras que el resto de los terminales, al estar formados por un solo carácter, pueden ser manipulados a través de su código ASCII. De esta manera, asumiendo que los espacios en blanco[2], tabuladores y retornos de carro actúan simplemente como separadores, disponemos de dos opciones principales:

```
%%      /* Opción 1 */
[0 - 9]+            { return NUM; }
[A- Z][A - Z0 -9]*  { return ID; }
[ \t\n]             { ; }
.                   { return yytext[0]; }     /* Error sintáctico. Le pasa el error al a.si. */
```

y

```
%%      /* Opción 2 */
```

[2] El espacio en blanco se representará mediante el carácter "␣".

```
[0 - 9]+              { return NUM; }
[A- Z][A - Z0 -9]*    { return ID; }
[ \t\n]               { ; }
'('|')'|'*'|'+'       { return yytext[0]; }     /* El a.si. sólo recibe tokens válidos */
.                     { printf ("Error léxico.\n"); }
```

Estas opciones se comportan de manera diferente ante los errores léxicos, como por ejemplo cuando el usuario teclea una '@' donde se esperaba un '+' o cualquier otro operador. En la Opción 1, el analizador sintáctico recibe el código ASCII de la '@' como *token*, debido al patrón **.**; en este caso el analizador se encuentra, por tanto, ante un *token* inesperado, lo que le lleva a abortar el análisis emitiendo un mensaje de error sintáctico. En la Opción 2 el analizador léxico protege al sintáctico encargándose él mismo de emitir los mensajes de error ante los lexemas inválidos; en este caso los mensajes de error se consideran lexicográficos; además, el desarrollador puede decidir si continuar o no el reconocimiento haciendo uso de la función **halt()** de C.

En los ejemplos que siguen nos decantamos por la Opción 1, ya que es el método del mínimo esfuerzo y concentra la gestión de errores en el analizador sintáctico.

Por último, nótese que el programa Lex no incluye la función **main()**, ya que PCYacc proporciona un modelo dirigido por sintaxis y es al analizador sintáctico al que se debe ceder el control principal, y no al léxico. Por tanto, será el área de funciones del programa Yacc quien incorpore en el **main()** una llamada al analizador sintáctico, a través de la función **yyparse()**:

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

4.3.1.3 Área de reglas

Éste es el área más importante ya que en él se especifican las reglas de producción que forman la gramática cuyo lenguaje queremos reconocer. Cada regla gramatical tienen la forma:

noTerminal: consecuente ;

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

El conjunto de reglas de producción de nuestro ejemplo se escribiría en Yacc tal y como se ilustra en el cuadro 4.5 donde, por convención, los símbolos no terminales se escriben en minúscula y los terminales en mayúscula, las flechas se sustituyen por ':' y, además, cada regla debe acabar en ';'

Gramática	Notación Yacc
	%% /* área de reglas */
E → E + T	e : e '+' t
| T	| t
	;
T → T * F	t : t '*' f
| F	| f
	;
F → M (E)	f : m '(' e ')'
| M id	| m ID
| M num	| m NUM
	;
M → ε	m : /* Épsilon */
| -	| '-'
	;

Cuadro 4.5 Correspondencia entre una gramática formal y su equivalente en notación Yacc.

Junto con los terminales y no terminales del consecuente, se pueden incluir acciones semánticas en forma de código en C delimitado por los símbolos **{** y **}**. Ejemplo:

```
e   :   e '+' t   {printf("Esto es una suma.\n");};
```

Si hay varias reglas con el mismo antecedente, es conveniente agruparlas indicando éste una sola vez y separando cada consecuente por el símbolo '|', con tan sólo un punto y coma al final del último consecuente. Por lo tanto las reglas gramaticales:

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

se pueden expresar en Yacc como:

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

4.3.1.4 Área de funciones

La tercera sección de una especificación en Yacc consta de rutinas de apoyo escritas en C. Aquí se debe proporcionar un analizador léxico mediante una función llamada **yylex()**. En caso necesario se pueden agregar otros procedimientos, como rutinas de apoyo o resumen de la compilación.

El analizador léxico **yylex()** produce pares formados por un *token* y su valor de atributo asociado. Si se devuelve un *token* como por ejemplo NUM, o ID, dicho *token* debe declararse en la primera sección de la especificación en Yacc, El valor del atributo asociado a un *token* se comunica al analizador sintáctico mediante la variable **yylval**, como se verá en el punto ?. En caso de integrar Yacc con Lex, en esta sección habrá de hacer un **#include** del fichero generado por PCLex.

En concreto, en esta sección se deben especificar como mínimo las funciones:

- **main()**, que deberá arrancar el analizador sintáctico haciendo una llamada a **yyparse()**.
- **void yyerror(char * s)** que recibe una cadena que describe el error. En este momento la variable **yychar** contiene el terminal que se ha recibido y que no se esperaba.

4.3.2 Gestión de atributos

Como ya se ha indicado, en una regla se pueden incluir acciones semánticas, que no son más que una secuencia de declaraciones locales y sentencias en lenguaje C. Pero la verdadera potencia de las acciones en Yacc es que a cada terminal o no terminal se le puede asociar un atributo. Los nombres de atributos son siempre los mismos y dependen de la posición que el símbolo ocupa dentro de la regla y no del símbolo en sí; de esta manera el nombre **$$** se refiere al atributo del antecedente de la regla, mientras que **$i** se refiere al atributo asociado al i-ésimo símbolo gramatical del consecuente, es decir:

- El atributo del antecedente es **$$**.
- El atributo del 1er símbolo del consecuente es **$1**.
- El atributo del 2º símbolo del consecuente es **$2**.
- Etc.

La acción semántica se ejecutará siempre que se reduzca por la regla de producción asociada, por lo que normalmente ésta se encarga de calcular un valor para $$ en función de los $i. Por ejemplo, si suponemos que el atributo de todos los símbolos de nuestra gramática (terminales y no terminales) es de tipo entero, se tendría:

```
%%
❶e  :   e '+' t     {$$ = $1 + $3;}
❷   |   t           {$$ = $1;}
    ;
❸t  :   t '*' f     { $$ = $1 * $3; }
❹   |   f           { $$ = $1; }
    ;
❺f  :   m '(' e ')'     { $$ = $1 * $3; }
❻   |   m ID            { $$ = $1 * $2; }
❼   |   m NUM           { $$ = $1 * $2; }
    ;
❽m  :   /* Épsilon */  { $$ = +1; }
❾   |   '-'             { $$ = -1; }
```

;

Por otro lado, en las reglas ❻ y ❼ del ejemplo anterior puede apreciarse que $2 se refiere a los atributos asociados a ID y NUM respectivamente; pero, al ser ID y NUM terminales, ¿qué valores tienen sus atributos?¿quién decide qué valores guardar en ellos?. La respuesta es que el analizador léxico es quien se tiene que encargar de asignar un valor a los atributos de los terminales. Recordemos que en el apartado 2.4.3.6 se estudió que PCLex proporcionaba la variable global **yylval** que permitía asociar información adicional a un *token*. Pues bien, mediante esta variable el analizador léxico puede comunicar atributos al sintáctico. De esta manera, cuando el analizador léxico recibe una cadena como "27+13+25", no sólo debe generar la secuencia de *tokens* NUM '+' NUM '+' NUM, sino que también debe cargar el valor del atributo de cada NUM, y debe de hacerlo antes de retornar el token (antes de hacer **return NUM;**). En nuestro caso asociaremos a cada NUM el valor entero que representa su lexema, con el objetivo de que las acciones semánticas del sintáctico puedan operar con él. El patrón Lex y la acción léxica asociada son:

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

La función **atoi()** convierte de texto a entero. Nótese que siempre que se invoque a **atoi()**, la conversión será correcta, ya que el lexema almacenado en **yytext** no puede contener más que dígitos merced al patrón asociado. Además, se asume que la variable **yylval** también es entera.

PCYacc necesita alguna información adicional para poder realizar una correcta gestión de atributos, es decir, hemos de informarle del tipo de atributo que tiene asociado cada símbolo de la gramática, ya sea terminal o no terminal. Para ello debe definirse la macro **YYSTYPE** que indica de qué tipo son todos los atributos; en otras palabras, PCYacc asume que todos los atributos son de tipo **YYSTYPE**, por lo que hay que darle un valor a esta macro. De esta forma, si queremos que todos los atributos sean de tipo **int**, bastaría con escribir

```
%{
#define YYSTYPE int
%}
```

como sección de declaraciones globales en el área de definiciones. De hecho, PCYacc obliga a incluir en los programas Yacc una declaración como ésta incluso aunque no se haga uso de los atributos para nada: PCYacc exige asociar atributos siempre y conocer su tipo.

No obstante, resulta evidente que no todos los símbolos de una gramática necesitarán atributos del mismo tipo, siendo muy común que los números tengan asociado un atributo entero, y que los identificadores posean un puntero a una tabla de símbolos que almacene información importante sobre él. En este caso, puede declararse **YYSTYPE** como una **union** de C de manera que, aunque todos los símbolos de la gramática poseerán como atributo dicha **union**, cada uno utilizará sólo el campo que precise. Aunque este método puede llegar a derrochar un poco de memoria, permite una gestión uniforme de la pila de análisis sintáctico, lo que conlleva una mayor eficiencia.

Es más, el mecanismo de usar una **union** como tipo de **YYSTYPE** es una práctica tan común que PCYacc posee una cláusula que permite definir **YYSTYPE** y declarar la **union** en un solo paso. La cláusula tiene la forma:

```
%union {
        /* Cuerpo de una union en C */
}
```

Recordemos que una **union** en C permite definir registros con parte variante. Es más, en una **union** todo es parte variante, o sea, todas los campos declarados en el cuerpo de una **union** comparten el mismo espacio de memoria, de manera que el tamaño de la **union** es el del campo más grande que contiene. Siguiendo con nuestro ejemplo, la unión tendría la forma:

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

Con esto, **YYSTYPE** se ha convertido en una **union** (ya no es necesario incluir el #define YYSTYPE), de manera que cualquier símbolo puede hacer referencia al campo **numero** o al campo **ptrNodo**, pero no a ambos. Por regla general, cada símbolo gramatical, independientemente de la regla en que aparezca utiliza siempre el mismo campo del **%union**; por ello, PCYacc suministra un mecanismo para indicar cuál de los campos es el que se pretende utilizar para cada símbolo. Con esto además, nos ahorramos continuas referencias a los campos del **%union**. El mecanismo consiste en incluir entre paréntesis angulares en la cláusula %**token** del área de definiciones el campo que se quiere utilizar para cada terminal de la gramática; para indicar el tipo de los atributos de los no terminales, se utilizará la cláusula **%type** que, por lo demás, es igual a la **%token** excepto porque en ella se enumeran no terminales en lugar de terminales. Siguiendo esta notación, nuestro ejemplo quedaría:

```
%union {
        int numero;
        nodo * ptrNodo;
}
%token <numero> NUM
%token <ptrNodo> ID
%type <numero> e t f m
%%
❶e  :   e '+' t     {$$ = $1 + $3;}
❷   |   t           {$$ = $1;}
    ;
❸t  :   t '*' f     { $$ = $1 * $3; }
❹   |   f           { $$ = $1; }
    ;
❺f  :   m '(' e ')'     { $$ = $1 * $3; }
❻   |   m ID            { $$ = /*Extraer alguna información de (*$2) */; }
❼   |   m NUM           { $$ = $1 * $2; }
    ;
```

```
❽m :    /* Épsilon */  { $$ = +1; }
❾  |    '-'            { $$ = -1; }
                ;
```

Nótese cómo, a pesar de que **YYSTYPE** es de tipo **union**, los $$ y $i no hacen referencia a sus campos, gracias a que las declaraciones **%token** y **%type** hacen estas referencias por nosotros. Es más, si en algún punto utilizamos el atributo de algún símbolo y no hemos especificado su tipo en una cláusula **%token** o **%type**, PCYacc emitirá un mensaje de error durante la metacompilación. Estas transformaciones son realizadas por PCYacc por lo que las acciones léxicas, gestionadas por PCLex, cuando hacen referencia a **yylval** deben indicar explícitamente el campo con que trabajar. El programa Lex que implementa el analizador léxico de nuestra gramática sería:

```
%%      /* Opción 1 */
[0 - 9]+            { yylval.numero = atoi(yytext); return NUM; }
[A- Z][A - Z0 -9]*  { yylval.ptrNodo = ...; return ID;}
[ \t\n]             { ; }
.                   { return yytext[0]; }
```

La gestión del atributo asociado a los identificadores de usuario la dejaremos para el capítulo dedicado a la tabla de símbolos. Por ahora nos centraremos sólo en las constantes numéricas. La figura ? muestra el árbol generado para reconocer la cadena "27+13+25"; los valores de los atributos aparecen como subíndices.

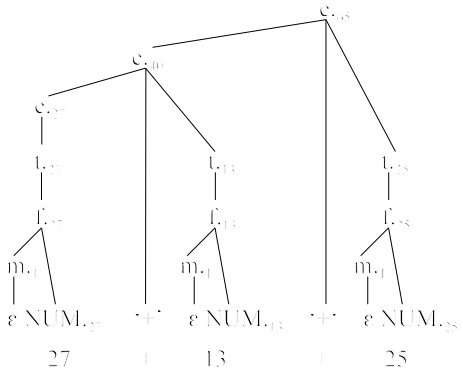


Figure 12 Árbol sintáctico para reconocer una cadena en la que no intervienen identificadores.

4.3.2.1 Acciones intermedias

También es posible colocar acciones en medio de una regla, esto es, entre los símbolos de un consecuente. Sin embargo, antes se dijo que PCYacc construye analizadores que ejecutan una acción semántica cada vez que se reduce por su regla asociada en la construcción de un árbol sintáctico. Entonces, ¿cuándo se ejecuta una acción intermedia? En estos casos, PCYacc realiza una conversión de la gramática proporciona a otra equivalente en la que la acción intermedia aparece a la derecha de una nueva regla de la forma : $N_i \rightarrow \varepsilon$, donde **i** es un número arbitrario generado por PCYacc. Por ejemplo,

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

se convierte a

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

La ejecución de acciones semánticas está sujeta al mecanismo visto en el punto 4.1.2.

4.3.3 Ambigüedad

PCYacc no sólo permite expresar de forma fácil una gramática a reconocer y generar su analizador sintáctico, sino que también da herramientas que facilitan la eliminación de ambigüedades.

La gramática propuesta en el apartado 4.3.1 es un artificio para obligar a que el producto y la división tenga más prioridad que la suma y la resta, y que el menos unario sea el operador de mayor prioridad. Sin embargo, desde el punto de vista exclusivamente sintáctico, el lenguaje generado por dicha gramática resulta mucho más fácil de expresar mediante la gramática

```
e       :       e '+' e
        |       e '-' e
        |       e '*' e
        |       e '/' e
        |       '-' e
        |       '(' e ')'
        |       ID
        |       NUM
        ;
```

que, como resulta evidente, es ambigua. Por ejemplo, ante la entrada "27*13+25" se pueden obtener los árboles sintácticos de la figura 4.13; en el caso a) primero se multiplica y luego se suma, mientras que en el b) primero se suma y luego se multiplica. Dado que el producto tiene prioridad sobre la suma, sólo el caso a) es válido desde el punto de vista semántico, pues de otro modo se obtiene un resultado incorrecto.

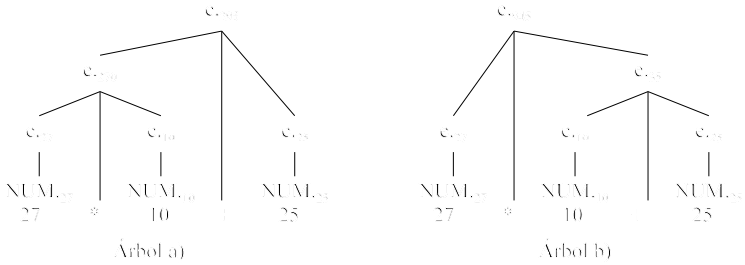


Figure 13Árboles sintácticos que reconocen una sentencia válida según una gramática ambigua. Nótese cómo semánticamente se producen resultados diferentes según el orden de evaluación. La opción correcta es la a), ya que el producto tiene prioridad sobre la suma.

Esta situación se produce porque cuando el analizador tiene en la pila expr.$_{27}$ '*' expr.$_{10}$ y se encuentra con el '+', se produce un conflicto desplazar/reducir: si se desplaza se obtiene el árbol b), y si se reduce se obtiene el árbol a). Este mismo problema se habría producido en caso de tener dos sumas, en lugar de un producto y una suma; análogamente un árbol como el a) habría sido el correcto, ya que la suma es asociativa por la izquierda. En general, si una regla es recursiva por la izquierda implica que el operador que contiene es asociativo por la izquierda; e igual por la derecha. Por eso la gramática propuesta es ambigua, ya que cada regla es simultáneamente recursiva por la derecha y por la izquierda lo que implica que los operadores son asociativos por la izquierda y por la derecha a la vez, lo que no tiene sentido. Aunque la mayoría de operadores son asociativos a la izquierda, también los hay a la derecha, como es el caso del operador de exponenciación. Es más, algunos operadores ni siquiera son asociativos como puede ser el operador de módulo (resto) de una división.

Aunque ya hemos visto que este problema se puede solucionar cambiando la gramática, PCYacc suministra un mecanismo para resolver las ambigüedades producidas por conflictos del tipo desplazar/reducir sin necesidad de introducir cambios en la gramática propuesta. A menos que se le ordene lo contrario, PCYacc resolverá todos los conflictos en la tabla de acciones del analizador sintáctico utilizando las dos premisas siguientes:

1. Un conflicto reducir/reducir se resuelve eligiendo la regla en conflicto que se haya escrito primero en el programa Yacc.
2. Un conflicto desplazar/reducir se resuelve en favor del desplazamiento.

Además de estas premisas, PCYacc proporciona al desarrollador la posibilidad de solucionar por sí mismo los conflictos de la gramática. La solución viene de la mano de las cláusulas de precedencia y asociatividad: **%left**, **%right**, **%nonassoc** y **%prec**.

Las cláusulas **%left**, **%right** y **%nonassoc** permiten indicar la asociatividad de un operador (*left*-izquierda: reducir, *right*-derecha: desplazar y *nonassoc*-no asociativo: error); además, el orden en que se especifiquen las cláusulas permite a PCYacc saber qué operadores tienen mayor prioridad sobre otros. Estas cláusulas se colocan en el área de definiciones, normalmente después de la lista de terminales. Por ejemplo, la cláusula

```
%left '-'
```

hace que la regla e : e '-' e deje de ser ambigua. Nótese que para PCYacc no existe el concepto de operador (aritmético o no), sino que en estas cláusulas se indica cualquier símbolo en que se produzca un conflicto desplazar/reducir, ya sea terminal o no terminal.

Si se escriben varios símbolos en un mismo **%left**, **%right** o **%nonassoc** estaremos indicando que todos tiene la misma prioridad. Una especificación 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 informa a PCYacc de que '*' y '/' tienen mayor prioridad que '+' y '-'.

Hay situaciones en las que un mismo terminal puede representar varias operaciones, como en el caso del signo '-' que puede ser el símbolo de la resta o del menos unario. Dado que sólo tiene sentido que el '-' aparezca en una única cláusula resulta imposible indicar que, cuando se trata del menos binario la prioridad es inferior a la del producto, mientras que si es unario la prioridad es superior. Para solucionar esta situación, PCYacc proporciona el modificador **%prec** que, colocado al lado del consecuente de una regla, modifica la prioridad y asociatividad de la misma. **%prec** debe ir acompañado de un operador previamente listado en alguna de las cláusulas **%left**, **%right** o **%nonassoc**, del cual hereda la precedencia y la prioridad. En nuestro caso de ejemplo se podría escribir:

```
%left '+' '-'
%left '*' '/'
%%
e       :       e '+' e
        |       e '-' e
        |       e '*' e
        |       e '/' e
        |       '-' e      %prec '*'
        |       '(' e ')'
        |       ID
        |       NUM
        ;
```

indicándose de esta forma que la regla del menos unario tiene tanta prioridad como la del producto y que, en caso de conflicto, también es asociativa por la izquierda. No

obstante, esta solución presenta dos problemas: 1) la prioridad del menos unario no es igual a la del producto, sino superior; y 2) el menos unario no es asociativo por la izquierda, sino por la derecha (aunque esto no daría problemas, ya que no existe ambigüedad en la formación del árbol asociado a una sentencia como "---5", tal y como ilustra la figura 4.14).

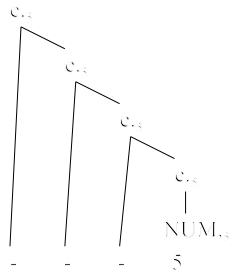


Figure 14 Aunque la gramática sea ambigua, si hay varios operadores "menos unario" seguidos, éstos sólo pueden asociarse a derecha.

Para resolver estos problemas, puede recurrirse a un artificio que consiste en declarar un terminal que representa a un operador ficticio de la prioridad y asociatividad requeridas con el único objetivo de usarlo posteriormente en un **%prec**. Este nuevo *token* no será retornado jamás por el analizador lexicográfico. Además, los terminales enumerados en una sentencia **%left**, **%right** o **%nonassoc** no hace falta declararlos también en el correspondiente **%token**, a no ser que posean un atributo del **%union**. Así, la solución final al problema de la prioridad y asociatividad viene dado por el bloque de programa Yacc:

```
%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO
%%
e       :       e '+' e
        |       e '-' e
        |       e '*' e
        |       e '/' e
        |       '-' e      %prec MENOS_UNARIO
        |       '(' e ')'
        |       ID
        |       NUM
        ;
```

4.3.4 Tratamiento de errores

Quizás una de las mayores deficiencias de PCYacc es la referente al tratamiento de errores ya que, por defecto, cuando el analizador sintáctico que genera se encuentra con un error sintáctico, sencillamente se para la ejecución con un parco mensaje "*syntax error*".

Esta ruptura de la ejecución puede evitarse mediante el correcto uso del *token* especial **error**. Básicamente, si se produce un error sintáctico el analizador generado por PCYacc desecha parte de la pila de análisis y parte de la entrada que aún le queda por leer, pero sólo hasta llegar a una situación donde el error se pueda recuperar, lo que se consigue a través del *token* reservado **error**. Un ejemplo típico del uso del *token* **error** es el siguiente:

```
prog    :       /* Épsilon */
        |       prog sent ';'
        |       prog error ';'
        ;
sent    :       ID '=' NUM
        ;
```

La última regla de este trozo de gramática sirve para indicarle a PCYacc que entre el no terminal **prog** y el terminal **';'** puede aparecer un error sintáctico. Supongamos que tenemos la sentencia a reconocer "a = 6; b = 7; c.= 8; d = 6;" en la que se ha "colado" un punto tras el identificador **c**; cuando el analizador se encuentre el *token* **'.'** se da cuenta de que hay un error sintáctico y busca en la gramática todas las reglas que contienen el símbolo **error**; a continuación va eliminando símbolos de la cima de la pila, hasta que los símbolos ubicados encima de la pila coincidan con los situados a la izquierda del símbolo **error** en alguna de las reglas en que éste aparece. A continuación va eliminando *tokens* de la entrada hasta encontrarse con uno que

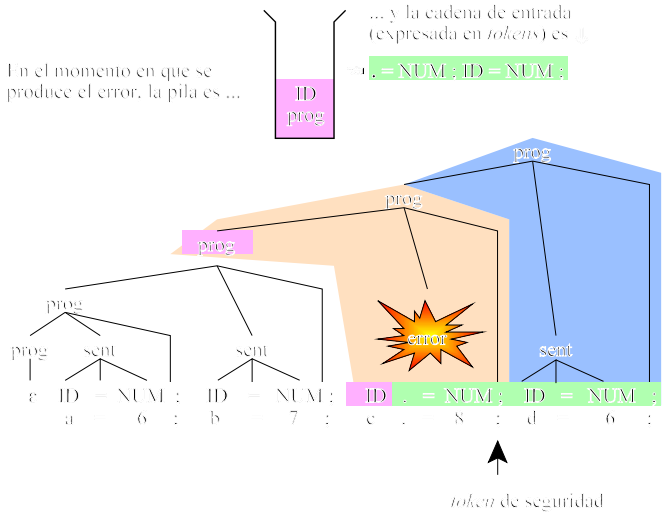


Figure 15 Recuperación de un error sintáctico. "Recuperar" no quiere decir "corregir", sino ser capaz de continuar el análisis sintáctico

coincida con el que hay justamente a la derecha en la regla de error seleccionada. Una vez hecho esto, inserta el *token* **error** en la pila, desplaza y continúa con el análisis intentando reducir por esa regla de error. Al terminal que hay justo a la derecha de **error** en una regla de error se le denomina *token* de seguridad, ya que permite al analizador llegar a una condición segura en el análisis. Como *token* de seguridad suele escogerse el de terminación de sentencia o de declaración, como en nuestro caso, en el que hemos escogido el punto y coma. La figura 4.15 ilustra gráficamente el proceso seguido para la recuperación del error.

Por último, para que todo funcione de forma adecuada es conveniente asociarle a la regla de error la invocación de una macro especial de PCYacc denominada **yyerrok**:

```
prog    :       /* Épsilon */
        |       prog sent ';'
        |       prog error ';'      { yyerrok; }
        ;
sent    :       ID '=' NUM
        ;
```

La ejecución de esta macro informa al analizador sintáctico de que la recuperación se ha realizado satisfactoriamente. Si no se invoca, el analizador generado por PCYacc entra en un estado llamado"condición de seguridad", y permanecerá en ese estado hasta haber desplazado tres *tokens*. La condición de seguridad previene al analizador de emitir más mensajes de error (ante nuevos errores sintácticos), y tiene por objetivo evitar errores en cascada debidos a una incorrecta recuperación del error. La invocación a **yyerrok** saca al analizador de la condición de seguridad, de manera que éste nos informará de todos, absolutamente todos, los errores sintácticos, incluso los debidos a una incorrecta recuperación de errores.

A pesar de poder recuperar los errores sintácticos, PCYacc no genera analizadores que informen correctamente del error, sino que solamente se limitan a informar de la existencia de éstos pero sin especificar su naturaleza. Esta falta se puede subsanar siguiendo los siguientes pasos:

1. En el área de funciones de Yacc, colocar la directiva:
   #include "errorlib.c"
   y no incluir la función **yyerror(char * s)**, que ya está codificada dentro del fichero **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 del analizador y el código asociado a cada *token*.
   -pyaccpar.c hace que se utilice un esqueleto para contener al autómata finito con pila que implementa PCYacc. 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 uso, a su vez, del fichero **errorlib.h**.

3. Ejecutar el programa
   tokens.com
   Este programa toma el fichero **yytab.h**, y genera el fichero **yytok.h** que no es más que la secuencia de nombres de *tokens* entrecomillada y separada por comas. Este fichero es utilizado por **errorlib.c** para indicar qué *tokens* se esperaban.

4. Compilar **fichero.c** (por ejemplo, con Turbo C):
   tc fichero.c

De esta manera, cuando se ejecute **fichero.exe** y se produzca un error sintáctico, se obtendrá un clarificador mensaje de error:
"Error en la línea ...: encontrado el token ... y esperaba uno de los tokens ..."
donde los puntos suspensivos son rellenados automáticamente por el analizador sintáctico con la línea del error, el *token* incorrecto que lo ha provocado y la lista de *tokens* que habrían sido válidos, respectivamente.

4.3.5 Ejemplo final

Por último, uniendo todos los bloques de código Yacc que hemos ido viendo en los epígrafes anteriores, el ejemplo de la calculadora quedaría:

```
%{
#define YYSTYPE int
%}
%token NUMERO
%left '+' '-'
%left '*' '/'
%right MENOS_UNARIO
%%
prog    :       /* Épsilon */
        |       prog expr ';'       { printf("= %d\n", $2); }
        |       prog error ';'      { yyerrok; }
        ;
expr    :       expr '+' expr       { $$ = $1 + $3; }
        |       expr '-' expr       { $$ = $1 - $3; }
        |       expr '*' expr       { $$ = $1 * $3; }
        |       expr '/' expr       { $$ = $1 / $3; }
        |       '-' expr   %prec MENOS_UNARIO   { $$ = - $2; }
        |       '(' expr ')'        { $$ = $2; }
        |       NUMERO              { $$ = $1; }
        ;
%%
#include "lexico.c"
void main() { yyparse(); }
void yyerror(char * s) { printf("%s\n", s); }
```

4.4 El generador de analizadores sintácticos Cup

Cup es una analizador sintáctico LALR desarrollado en el Instituto de Tecnología de Georgia (EE.UU.) que genera código Java y que permite introducir acciones semánticas escritas en dicho lenguaje. Utiliza una notación bastante parecida a la de PCYacc e igualmente basada en reglas de producción.

4.4.1 Diferencias principales

A modo de introducción, las principales diferencias con PCYacc, son:

- El antecedente de una regla se separa del consecuente mediante ::=, en lugar de mediante :.
- No es posible utilizar caracteres simples (´+´, ´-´, ´,´etc.) como terminales en una regla de producción, de manera que todos los terminales deben declararse explícitamente: MAS, MENOS, COMA, etc.
- En lugar de **%token** utiliza terminal.
- En lugar de **%type** utiliza non terminal.
- La lista de terminales se debe separar con comas y finalizar en punto y coma, al igual que la de no terminales. En ambas listas deben aparecer todos los terminales y no terminales, tanto si tienen atributo como si no.
- Si un terminal o no terminal tiene asociado un atributo, éste no puede ser de tipo primitivo sino que, obligatoriamente, debe ser un objeto (**Object**). Por tanto, en lugar de usar **int** se deberá usar **Integer**, en lugar de **char** se deberá usar **Character**, etc.
- No existe el **%union**, sino que se indica directamente el tipo de cada terminal y no terminal mediante el nombre de la clase correspondiente: **Integer**, **Character**, **MiRegistro**, etc. Tampoco es necesario el uso de corchetes angulares para indicar este tipo.
- En las acciones semánticas intermedias, los atributos del consecuente que preceden a la acción pueden usarse en modo sólo lectura (como si fueran del tipo **final** de Java).
- En lugar de **%left**, **%right** y **%nonassoc** se emplean, respectivamente precedence left, precedence right y precedence nonassoc. No obstante, la cláusula **%prec** sí se utiliza igual que en PCYacc.
- En lugar de **%start** se utiliza **start with**.
- No existen secciones que se deban separar por los símbolos %%.
- Pueden utilizarse, como si de un programa Java se tratase, las cláusulas **import** y **package**.

- Antes de empezar la ejecución del analizador generado puede ejecutarse un bloque de código Java mediante la cláusula:
  init with {: bloque_de_código_Java :};
- Para conectar el analizar generado con el analizador léxicográfico se utiliza la cláusula:
  scan with {: bloque_Java_que_retorna_un_objeto_de_la_clase_sym :};

Con estas pequeñas diferencias podemos ver a continuación cómo quedaría la especificación de nuestra calculadora:

```
package primero;
import java_cup.runtime.*;
/* Inicialización del analizador léxico (si es que hiciera falta) */
init with {: scanner.init(); :};
/* Solicitud de un terminal al analizador léxico. */
scan with {: return scanner.next_token(); :};
/* Terminales  */
terminal PUNTOYCOMA, MAS, MENOS, POR, ENTRE, MODULO;
terminal UMENOS, LPAREN, RPAREN;
terminal Integer NUMERO;
/* No terminales */
non terminal expr_list, expr_part;
non terminal Integer expr, term, factor;
/* Precedencias */
precedence left MAS, MENOS;
precedence left POR, ENTRE, MODULO;
precedence left UMENOS;
/* Gramática*/
lista_expr          ::=     lista_expr expr PUNTOYCOMA
                    |       /* Epsilon */
                    ;
expr        ::=     expr MAS expr
            |       expr MENOS expr
            |       expr POR expr
            |       expr ENTRE expr
            |       expr MODULO expr
            |       MENOS expr %prec UMENOS
            |       LPAREN expr RPAREN
            |       NUMERO
            ;
```

El propio Cup está escrito en Java, por lo que, para su ejecución, es necesario tener instalado algún JRE (*Java Runtime Environment*). La aplicación principal viene representada por la clase **java_cup.Main**, de tal manera que si el ejemplo de la calculado está almacenado en un fichero llamado **calculadora.cup**, se compilaría de la forma:

```
java java_cup.Main calculadora.cup
```

lo que produce los ficheros **sym.java** y **parser.java**. Como puede observarse, Cup no

respeta la convención (ampliamente aceptada) de nombrar con mayúsculas a las clases. La clase **sym.java** contiene las constantes necesarias para poder referirse a los terminales: sym.MAS, sym.POR, etc., lo que facilita el trabajo al analizador lexicográfico, que deberá retornar dichos terminales mediante sentencias de la forma: return new Symbol(sym.MAS); por ejemplo.

Para continuar nuestro ejemplo de la calculadora, es necesario incluir acciones semánticas y poder realizar referencias a los atributos tanto de los símbolos del consecuente, como del no terminal antecedente. De manera parecida a como sucede con Yacc, las acciones semánticas vienen encapsuladas entre los caracteres **{:** y **:}**. Sin embargo, la forma de referir los atributos difiere notablemente al método usado por Yacc.

Para empezar, el atributo del antecedente no se llama **$$**, sino **RESULT**; de manera similar, los atributos del consecuente no van numerados sino nombrados, lo que quiere decir que es el propio desarrollador quien debe decidir qué nombre se le va a dar al atributo de cada símbolo del consecuente. Este nombre se le indica a Cup colocándolo a continuación del terminal/no terminal en la regla de producción y separado de éste mediante el carácter dos puntos. De esta manera, a una regla como:

```
expr    ::=     expr MAS expr ;
```

se le daría significado semántico de la forma:

```
expr    ::=     expr:eIzq MAS expr:eDch
                {:RESULT = new Integer(eIzq.intValue() +
                                        eDch.intValue()); :}
        ;
```

Siguiendo este mecanismo, añadiendo el resto de acciones semánticas a nuestro ejemplo de la calculadora se obtiene:

```
import java_cup.runtime.*;
/* Inicialización del analizador léxico (si es que hiciera falta) */
init with {: scanner.init(); :};
/* Solicitud de un terminal al analizador léxico */
scan with {: return scanner.next_token(); :};
/* Terminales sin atributo */
terminal PUNTOYCOMA, MAS, MENOS, POR, ENTRE, MODULO;
terminal UMENOS, LPAREN, RPAREN;
/* Terminales con atributo asociado */
terminal Integer NUMERO;
/* No terminales sin atributo */
non terminal lista_expr;
/* No terminales con atributo asociado */
non terminal Integer expr;
/* Precedencias */
precedence left MAS, MENOS;
precedence left POR, ENTRE, MODULO;
precedence left UMENOS;
```

```
/* Gramática */
lista_expr    ::=    lista_expr expr:e PUNTOYCOMA  {: System.out.println("= " + e); :}
                |    lista_expr error PUNTOYCOMA
                |    /* Epsilon */
                ;
expr ::= expr:e1 MAS expr:e2
                    {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
        |    expr:e1 MENOS expr:e2
                    {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
        |    expr:e1 POR expr:e2
                    {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
        |    expr:e1 ENTRE expr:e2
                    {: RESULT = new Integer(e1.intValue() / e2.intValue()); :}
        |    expr:e1 MODULO expr:e2
                    {: RESULT = new Integer(e1.intValue() % e2.intValue()); :}
        |    NUMERO:n
                    {: RESULT = n; :}
        |    MENOS expr:e %prec UMENOS
                    {: RESULT = new Integer(0 - e.intValue()); :}
        |    LPAREN expr:e RPAREN
                    {: RESULT = e; :}
        ;
```

Como se ha dejado entrever anteriormente, para que todo esto funcione es necesario interconectarlo con un analizador lexicográfico que, ante cada terminal devuelva un objeto de tipo **java_cup.runtime.Symbol**. Los objetos de esta clase poseen un campo **value** de tipo **Object** que representa a su atributo, y al que el analizador lexicográfico debe asignarle un objeto coherente con la clase especificada en la declaración de terminales hecha en Cup.

4.4.2 Sintaxis completa.

Un fichero Cup está formado por cinco bloques principales que veremos a continuación.

4.4.2.1 Especificación del paquete e importaciones.

Bajo este primer bloque se indican opcionalmente las clases que se necesitan importar. También puede indicarse el paquete al que se quieren hacer pertenecer las clases generadas.

4.4.2.2 Código de usuario.

Cup engloba en una clase no pública aparte (incluida dentro del fichero **parser.java**) a todas las acciones semánticas especificadas por el usuario. Si el desarrollador lo desea puede incluir un bloque java dentro de esta clase mediante la declaración:

action code {: bloque_java :}

En este bloque pueden declararse variables, funciones, etc. todas de tipo

estático ya que no existen objetos accesibles mediante los que referenciar componentes no estáticos. Todo lo que aquí se declare será accesible a las acciones semánticas.

También es posible realizar declaraciones Java dentro de la propia clase **parser**, mediante la declaración:

parser code {: bloque_java :}

lo cual es muy útil para incluir el método **main()** que arranque nuestra aplicación.

Las siguientes dos declaraciones sirven para realizar la comunicación con el analizador léxicográfico. La primera es:

init with {: bloque_java :}

y ejecuta el código indicado justo antes de realizar la solicitud del primer *token*; el objetivo puede ser abrir un fichero, inicializar estructuras de almacenamiento, etc.

Por último, Cup puede comunicarse con cualquier analizador léxico, ya que la declaración:

scan with {: bloque_java :}

permite especificar el bloque de código que devuelve el siguiente *token* a la entrada.

4.4.2.3 Listas de símbolos

En este apartado se enumeran todos los terminales y no terminales de la gramática. Pueden tenerse varias listas de terminales y no terminales, siempre y cuando éstos no se repitan. Cada lista de terminales se hace preceder de la palabra **terminal**, y cada lista de no terminales se precede de las palabras **non terminal**. Los elementos de ambas listas deben estar separados por comas, finalizando ésta en punto y coma.

Con estas listas también es posible especificar el atributo de cada símbolo, ya sea terminal o no, sabiendo que éstos deben heredar de **Object** forzosamente, es decir, no pueden ser valores primitivos. Para ello, los símbolos cuyo tipo de atributo sea coincidente se pueden agrupar en una misma lista a la que se asociará dicho tipo justo detrás de la palabra **terminal**, de la forma:

terminal NombreClase terminal1, terminal2, etc.;

o bien

non terminal NombreClase noTerminal1, noTerminal2, etc.;

4.4.2.4 Asociatividad y precedencia.

Este apartado tiene la misma semántica que su homólogo en PCYacc, esto es, permite resolver los conflictos desplazar/reducir ante determinados terminales. De esta manera:

precedence left terminal1, terminal2, etc.;

opta por reducir en vez de desplazar al encontrarse un conflicto en el que el siguiente *token* es terminal1 o terminal2, etc. Por otro lado:

precedence right terminal1, terminal2, etc.;

opta por desplazar en los mismos casos. Finalmente:

precedence nonassoc terminal1, terminal2, etc.;

produciría un error sintáctico en caso de encontrarse con un conflicto desplazar/reducir en tiempo de análisis.

Pueden indicarse tantas cláusulas de este tipo como se consideren necesarias, sabiendo que el orden en el que aparezcan hace que se reduzca primero al encontrar los terminales de la últimas listas, esto es, mientras más abajo se encuentre una cláusula de precedencia, más prioridad tendrá a la hora de reducir por ella. De esta manera, es normal encontrarse cosas como:

precedence left SUMAR, RESTAR;
precedence left MULTIPLICAR, DIVIDIR;

lo que quiere decir que, en caso de ambigüedad, MULTIPLICAR y DIVIDIR tiene más prioridad que SUMAR y RESTAR, y que todas las operaciones son asociativas a la izquierda. Este tipo de construcciones sólo tiene sentido en gramáticas ambiguas, como pueda ser:

```
expr    ::=     expr MAS expr
            |   expr MENOS expr
            |   expr POR expr
            |   expr ENTRE expr
            |   LPAREN expr RPAREN
            |   NUMERO
            ;
```

Se asume que los símbolos que no aparecen en ninguna de estas listas poseen la prioridad más baja.

4.4.2.5 Gramática.

El último apartado de un programa Cup es la gramática a reconocer, expresada mediante reglas de producción que deben acabar en punto y coma, y donde el símbolo **::=** hace las veces de flecha.

La gramática puede comenzar con una declaración que diga cuál es el axioma inicial. Caso de omitirse esta cláusula se asume que el axioma inicial es el antecedente de la primera regla. Dicha declaración se hace de la forma:

start with noTerminal;

Al igual que en Yacc, es posible cambiar la prioridad y precedencia de una regla que produzca conflicto desplazar/reducir indicando la cláusula:

%prec terminal

junto a ella.

Las reglas de producción permiten albergar en su interior acciones semánticas, que son delimitadas por los símbolos **{:** y **:}**. En estas acciones puede colocarse cualquier bloque de código Java, siendo de especial importancia los accesos a los atributos de los símbolos de la regla. El atributo del antecedente viene dado por la variable Java **RESULT**, mientras que los atributos de los símbolos del consecuente son accesibles mediante los nombres que el usuario haya especificado para cada uno de

ellos. Este nombre se indica a la derecha del símbolo en cuestión y separado de éste por dos puntos. Su utilización dentro de la acción semántica debe ser coherente con el tipo asignado al símbolo cuando se lo indicó en la lista de símbolos. Debe prestarse especial atención cuando se utilicen acciones semánticas intermedias ya que, en tal caso, los atributos de los símbolos accesibles del consecuente sólo podrán usarse en modo lectura, lo que suele obligar a darles un valor inicial desde Jflex. Un ejemplo de esta situación podrá estudiarse en el apartado 8.4.6.

Por otro lado, y al igual que ocurría con PCYacc, el símbolo especial **error** permite la recuperación de errores mediante el método *panic mode*. En el apartado 7.4.4 se ilustra con más detalle la adecuada gestión de errores en Cup.

4.4.3 Ejecución de Cup.

El propio metacompilador Cup está escrito en Java, por lo que su ejecución obedece al modelo propio de cualquier programa Java. La clase principal es **java_cup.Main**, y toma su entrada de la entrada estándar:

```
java java_cup.Main < ficheroEntrada
```

Tambien es posible especificar algunas opciones al metacompilador, de la forma:

```
java java_cup.Main opciones < ficheroEntrada
```

donde las opciones más interesantes son:

-package nombrePaquete: las clases **parser** y **sym** se ubicarán en el paquete indicado.

- parser nombreParser: el analizador sintáctico se llamará **nombreParser** y no **parser**.

- symbols nombreSímbolos: la clase que agrupa a los símbolos de la gramática se llamará **nombreSímbolos** y no **sym**.

- expect numero: por defecto, Cup aborta la metacompilación cuando se encuentra con un conflicto reducir/reducir. Si se desea que el comportamiento se idéntico al de PCYacc, esto es, escoger la primera regla en orden de aparición en caso de dicho conflicto, es necesario indicarle a Cup exactamente el número de conflictos reducir/reducir que se esperan. Ello se indica mediante esta opción.

-progress: hace que Cup vaya informando a medida que va procesando la gramática de entrada.

-dump: produce un volcado legible de la gramática, los estados del autómata generado por Cup, y de las tablas de análisis. si se desea obtener sólo las tablas de análisis (utilizadas para resolver los conflictos), puede hacerse uso de la opción más concisa **-dump_states** que omite la gramática y las tablas de análisis.

Por otro lado, para que Cup se comunique convenientemente con el analizador léxico, éste debe implementar la interface **java_cup.runtime.Scanner**, definida como:

```
public interface Scanner {
        public Symbol next_token() throws java.lang.Exception;
}
```

Por último, para que nuestra aplicación funcione es necesario incluir una función **main()** de Java que construya un objeto de tipo **parser**, le asocie un objeto de tipo **Scanner** (que realizará el análisis lexicográfico), y arranque el proceso invocando a la función **parser()** dentro de una sentencia **try-catch**. Esto se puede hacer de la forma:

```
parser code {:
    public static void main(String[] arg){
        /* Crea un objeto parser */
        parser parserObj = new parser();
        /* Asigna el Scanner */
        Scanner miAnalizadorLexico =
                new Yylex(new InputStreamReader(System.in));
        parserObj.setScanner(miAnalizadorLexico);
      try{
            parserObj.parse();
        }catch(Exception x){
            System.out.println("Error fatal.");
        }
    }
:};
```

suponiendo que **Yylex** es la clase que implementa al analizador léxico.

4.4.4 Comunicación con JFlex

JFlex incorpora la opción **%cup** que equivale a las cláusulas:

```
%implements java_cup.runtime.scanner
%function next_token
%type java_cup.runtime.Symbol
%eofval{
        return new java_cup.runtime.Symbol(sym.EOF);
%eofval}
%eofclose
```

Si el usuario ha cambiado el nombre de la clase que agrupa a los símbolos (por defecto **sym**), también deberá utilizar en JFlex la directiva:

```
%cupsym nombreSímbolos
```

para que **%cup** genere el código correctamente.

Como se desprende de las cláusulas anteriores, el analizador lexicográfico debe devolver los *tokens* al sintáctico en forma de objetos de la clase **Symbol**. Esta clase, que se encuentra en el paquete **java_cup.runtime** y que, por tanto, debe ser importada por el lexicográfico, posee dos constructores:

- Symbol(int n). Haciendo uso de las constantes enteras definidas en la clase

sym, este constructor permite crear objetos sin atributo asociado.

- Symbol(int n, Object o). Análogo al anterior, pero con un parámetro adicional que constituye el valor del atributo asociado al *token* a retornar.

De esta manera, el analizador léxico necesario para reconocer la gramática Cup que implementa nuestra calculadora, vendría dado por el siguiente programa JFlex:

```
import java_cup.runtime.*;
%%
%unicode
%cup
%line
%column
%%
"+"                 { return new Symbol(sym.MAS); }
"-"                 { return new Symbol(sym.MENOS); }
"*"                 { return new Symbol(sym.POR); }
"/"                 { return new Symbol(sym.ENTRE); }
"%"                 { return new Symbol(sym.MODULO); }
";"                 { return new Symbol(sym.PUNTOYCOMA); }
"("                 { return new Symbol(sym.LPAREN); }
")"                 { return new Symbol(sym.RPAREN); }
[:digit:]+          { return new Symbol(sym.NUMERO, new Integer(yytext())); }
[ \t\r\n]+  {;}
.                   { System.out.println("Error léxico."+yytext()+"-"); }
```