Capítulo 3

Análisis sintáctico

3.1 Visión general

Todo lenguaje de programación obedece a unas reglas que describen la estructura sintáctica de los programas bien formados que acepta. En Pascal, por ejemplo, un programa se compone de bloques; un bloque, de sentencias; una sentencia, de expresiones; una expresión, de componentes léxicos; y así sucesivamente hasta llegar a los caracteres básicos. Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas de contexto libre o utilizando notación BNF (*Backus-Naur Form*).

Las gramáticas formales ofrecen ventajas significativas a los diseñadores de lenguajes y a los desarrolladores de compiladores:

- Las gramáticas son especificaciones sintácticas y precisas de lenguajes de programación.
- A partir de una gramática se puede generar automáticamente un analizador sintáctico.
- El proceso de generación automática anterior puede llevar a descubrir ambigüedades.
- Una gramática proporciona una estructura a un lenguaje de programación, siendo más fácil generar código y detectar errores.
- Es más fácil ampliar/modificar el lenguaje si está descrito con una gramática.

La mayor parte del presente tema está dedicada a los métodos de análisis sintáctico de uso típico en compiladores. Comenzaremos con los conceptos básicos y las técnicas adecuadas para la aplicación manual. A continuación trataremos la gestión y recuperación de errores sintácticos. Estudiaremos los métodos formales de análisis mediante autómatas con pila y, por último, profundizaremos en el funcionamiento de metacompiladores que generan automáticamente el analizador sintáctico de un lenguaje.

3.2 Concepto de analizador sintáctico

Es la fase del analizador que se encarga de chequear la secuencia de *tokens* que representa al texto de entrada, en base a una gramática dada. En caso de que el programa de entrada sea válido, suministra el árbol sintáctico que lo reconoce en base a una representación computacional. Este árbol es el punto de partida de la fase posterior de la etapa de análisis: el analizador semántico.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
BY NC SA

Pero esto es la teoría; en la práctica, el analizador sintáctico dirige el proceso de compilación, de manera que el resto de fases evolucionan a medida que el sintáctico va reconociendo la secuencia de entrada por lo que, a menudo, el árbol ni siquiera se genera realmente.

En la práctica, el analizador sintáctico también:

- Incorpora acciones semánticas en las que colocar el resto de fases del compilador (excepto el analizador léxico): desde el análisis semántico hasta la generación de código.
- Informa de la naturaleza de los errores sintácticos que encuentra e intenta recuperarse de ellos para continuar la compilación.
- Controla el flujo de *tokens* reconocidos por parte del analizador léxico.

En definitiva, realiza casi todas las operaciones de la compilación, dando lugar a un método de trabajo denominado **compilación dirigida por sintaxis**.

3.3 Manejo de errores sintácticos

Si un compilador tuviera que procesar sólo programas correctos, su diseño e implementación se simplificarían mucho. Las primeras versiones de los programas suelen ser incorrectas, y un buen compilador debería ayudar al programador a identificar y localizar errores. Es más, considerar desde el principio el manejo de errores puede simplificar la estructura de un compilador y mejorar su respuesta a los errores.

Los errores en la programación pueden ser de los siguientes tipos:

- Léxicos, producidos al escribir mal un identificador, una palabra clave o un operador.
- Sintácticos, por una expresión aritmética o paréntesis no equilibrados.
- Semánticos, como un operador aplicado a un operando incompatible.
- Lógicos, puede ser una llamada infinitamente recursiva.
- De corrección, cuando el programa no hace lo que el programador realmente deseaba.

Resulta evidente que los errores de corrección no pueden ser detectados por un compilador, ya que en ellos interviene el concepto abstracto que el programador tiene sobre el programa que construye, lo cual es desconocido, y probablemente incognoscible, por el compilador. Por otro lado, la detección de errores lógicos implica un esfuerzo computacional muy grande en tanto que el compilador debe ser capaz de averiguar los distintos flujos que puede seguir un programa en ejecución lo cual, en muchos casos, no sólo es costoso, sino también imposible. Por todo esto, los compiladores actuales se centran en el reconocimiento de los tres primeros tipos de errores. En este tema hablaremos de los errores de sintaxis, que son los que pueden impedir la correcta construcción de un árbol sintáctico.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

El manejo de errores de sintaxis es el más complicado desde el punto de vista de la creación de compiladores. Nos interesa que cuando el compilador encuentre un error, no cancele definitivamente la compilación, sino que se recupere y siga buscando errores. Recuperar un error no quiere decir corregirlo, sino ser capaz de seguir construyendo el árbol sintáctico a pesar de los errores encontrados. En vista de esto, el manejador de errores de un analizador sintáctico debe tener como objetivos:

- Indicar los errores de forma clara y precisa. Debe informar mediante los correspondientes mensajes del tipo de error y su localización.
- Recuperarse del error, para poder seguir examinando la entrada.
- Distinguir entre errores y advertencias. Las advertencias se suelen utilizar para informar sobre sentencias válidas pero que, por ser poco frecuentes, pueden constituir una fuente de errores lógicos.
- No ralentizar significativamente la compilación.

Un buen compilador debe hacerse siempre teniendo también en mente los errores que se pueden producir, con lo que se consigue simplificar su estructura. Además, si el propio compilador está preparado para admitir incluso los errores más frecuentes, entonces se puede mejorar la respuesta ante esos errores incluso corrigiéndolos.

A continuación se estudiarán varias estrategias para gestionar los errores una vez detectados

3.3.1 Ignorar el problema

Esta estrategia (denominada *panic mode* en inglés) consiste en ignorar el resto de la entrada hasta llegar a una condición de seguridad. Una condición tal se produce cuando nos encontramos un *token* especial (por ejemplo un ';' o un 'END'). A partir de este punto se sigue analizando normalmente. Los *tokens* encontrados desde la detección del error hasta la condición del error son desechados, así como la secuencia de *tokens* previa al error que se estime oportuna (normalmente hasta la anterior condición de seguridad).

Por ejemplo, supongamos la siguiente secuencia de sentencias de asignación en C o Java:

```
aux = a[i]
a[i] = a[j];
a[j] = aux;
```

La secuencia que se encuentra el analizador sintáctico aparece en la figura 3.1; claramente, falta el punto y coma de la primera sentencia y, dado que no tiene sentido una sentencia como:

```
aux = a[i] a[i] = a[j];
```

pues el analizador sintáctico advierte un error, y aprovecha que el ';' se usa como terminador de sentencias para ignorar el resto de la entrada hasta el siguiente ';'. Algunas consideraciones adicionales sobre esta estrategia de gestión de errores pueden estudiarse en los puntos 4.3.4 y 8.4.1.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

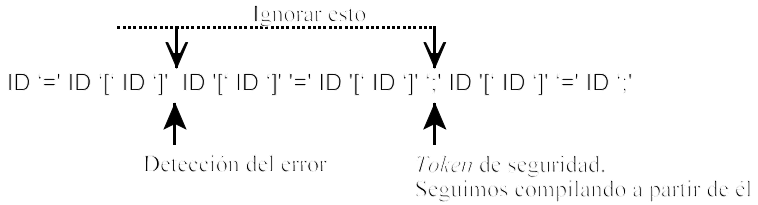


Figure 1 Recuperación de error sintáctico ignorando la secuencia errónea

3.3.2 Recuperación a nivel de frase

Intenta corregir el error una vez descubierto. P.ej., en el caso propuesto en el punto anterior, podría haber sido lo suficientemente inteligente como para insertar el *token* ';' . Hay que tener cuidado con este método, pues puede dar lugar a recuperaciones infinitas, esto es, situaciones en las que el intento de corrección no es el acertado, sino que introduce un nuevo error que, a su vez, se intenta corregir de la misma manera equivocada y así sucesivamente.

3.3.3 Reglas de producción adicionales

Este mecanismo añade a la gramática formal que describe el lenguaje reglas de producción para reconocer los errores más comunes. Siguiendo con el caso del punto 3.3.1 , se podría haber puesto algo como:

sent_errónea → sent_sin_acabar sent_acabada
sent_acabada → sentencia ';'
sent_sin_acabar → sentencia

lo cual nos da mayor control, e incluso permite recuperar y corregir el problema y emitir un mensaje de advertencia en lugar de uno de error.

3.3.4 Corrección Global

Este método trata por todos los medios de obtener un árbol sintáctico para una secuencia de *tokens*. Si hay algún error y la secuencia no se puede reconocer, entonces este método infiere una secuencia de *tokens* sintácticamente correcta lo más parecida a la original y genera el árbol para dicha secuencia. Es decir, el analizador sintáctico le pide toda la secuencia de *tokens* al léxico, y lo que hace es devolver lo más parecido a la cadena de entrada pero sin errores, así como el árbol que lo reconoce.

3.4 Gramática utilizada por un analizador sintáctico

Centraremos nuestros esfuerzos en el estudio de análisis sintácticos para lenguajes basados en gramáticas formales, ya que de otra forma se hace muy difícil la comprensión del compilador. La formalización del lenguaje que acepta un compilador sistematiza su construcción y permite corregir, quizás más fácilmente, errores de muy

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

difícil localización, como es la ambigüedad en el reconocimiento de ciertas sentencias.

La gramática que acepta el analizador sintáctico es una gramática de contexto libre, puesto que no es fácil comprender gramáticas más complejas ni construir automáticamente autómatas reducidos que reconozcan las sentencias que aceptan. Recuérdese que una gramática G queda definida por una tupla de cuatro elementos (N, T, P, S), donde:

N = No terminales.
T = Terminales.
P = Reglas de Producción.
S = Axioma Inicial.

Por ejemplo, una gramática no ambigua que reconoce las operaciones aritméticas podría ser la del cuadro 3.1. En ésta se tiene que N = {E, T, F} ya que E, T y F aparecen a la izquierda de alguna regla; T = {id, num, (,) ,+ ,*}; P son las siete reglas de producción, y S = E, pues por defecto el axioma inicial es el antecedente de la primera regla de producción.

① E	→	E + T
②	|	T
③ T	→	T * F
④	|	F
⑤ F	→	id
⑥	|	num
⑦	|	(E)

Cuadro 3.1 Gramática no ambigua que reconoce expresiones aritméticas

3.4.1 Derivaciones

Una regla de producción puede considerarse como equivalente a una regla de reescritura, donde el no terminal de la izquierda es sustituido por la pseudocadena del lado derecho de la producción. Podemos considerar que una pseudocadena es cualquier secuencia de terminales y/o no terminales. Formalmente, se dice que una pseudocadena α deriva en una pseudocadena β, y se denota por $\alpha \Rightarrow \beta$, cuando:

$$\alpha \Rightarrow \beta \text{ con } \alpha, \beta \in (N \cup T)^*, \text{ y siendo:}$$
$$\left.\begin{array}{l}\alpha \equiv \sigma\ A\ \delta \\ \beta \equiv \sigma\ \tau\ \delta\end{array}\right\} \text{ donde } A \in N \wedge \sigma, \tau, \delta \in (N \cup T)^* \wedge A \rightarrow \tau \in P$$

Nótese que dentro de un ‘α’ puede haber varios no terminales ‘A’ que pueden ser reescritos, lo que da lugar a varias derivaciones posibles. Esto hace que aparezcan los siguientes conceptos:

- Derivación por la izquierda: es aquella en la que la reescritura se realiza sobre el no terminal más a la izquierda de la pseudocadena de partida.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

- Derivación por la derecha: es aquella en la que la reescritura se realiza sobre el no terminal más a la derecha de la pseudocadena de partida.

Por ejemplo, partiendo de la gramática del cuadro 3.2, vamos a realizar todas las derivaciones a derecha e izquierda que podamos, a partir del axioma inicial, y teniendo como objetivo construir la cadena de *tokens*: (id_1 * id_2 + id_3). En cada pseudocadena aparece apuntado por una flechita el no terminal que se escoge para realizar la siguiente derivación.

① E	→	E + E
②	|	E * E
③	|	num
④	|	id
⑤	|	(E)

Cuadro 3.2 Gramática ambigua que reconoce expresiones aritméticas

Las derivaciones a izquierda quedarían:

$$E \Rightarrow E + E \Rightarrow E * E + E \Rightarrow id_1 * E + E \Rightarrow id_1 * id_2 + E \Rightarrow id_1 * id_2 + id_3$$

Mientras que las de a derecha darían lugar a:

$$E \Rightarrow E + E \Rightarrow E + id_3 \Rightarrow E * E + id_3 \Rightarrow E * id_2 + id_3 \Rightarrow id_1 * id_2 + id_3$$

Las pseudocadenas que se originan tras una secuencia cualquiera de derivaciones a partir del axioma inicial reciben el nombre de **formas sentenciales**.

3.4.2 Árbol sintáctico de una sentencia de un lenguaje

Al igual que podemos analizar una sentencia en español y obtener su árbol sintáctico, también es posible hacerlo con una sentencia de un lenguaje de programación. Básicamente un árbol sintáctico se corresponde con una sentencia, obedece a una gramática, y constituye una representación que se utiliza para describir el proceso de derivación de dicha sentencia. La raíz del árbol es el axioma inicial y, según nos convenga, lo dibujaremos en la cima o en el fondo del árbol.

Como nodos internos del árbol, se sitúan los elementos no terminales de las reglas de producción que vayamos aplicando, y cada uno de ellos poseerá tantos hijos como símbolos existan en la parte derecha de la regla aplicada.

Veamos un ejemplo utilizando la gramática del cuadro 3.1; supongamos que hay que reconocer la cadena (a + b) * a + b que el analizador léxico nos ha suministrado como (id_a + id_b) * id_a + id_b y vamos a construir el árbol sintáctico, que sería el de la figura 3.2.

Nótese que una secuencia de derivaciones con origen en el axioma inicial y

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

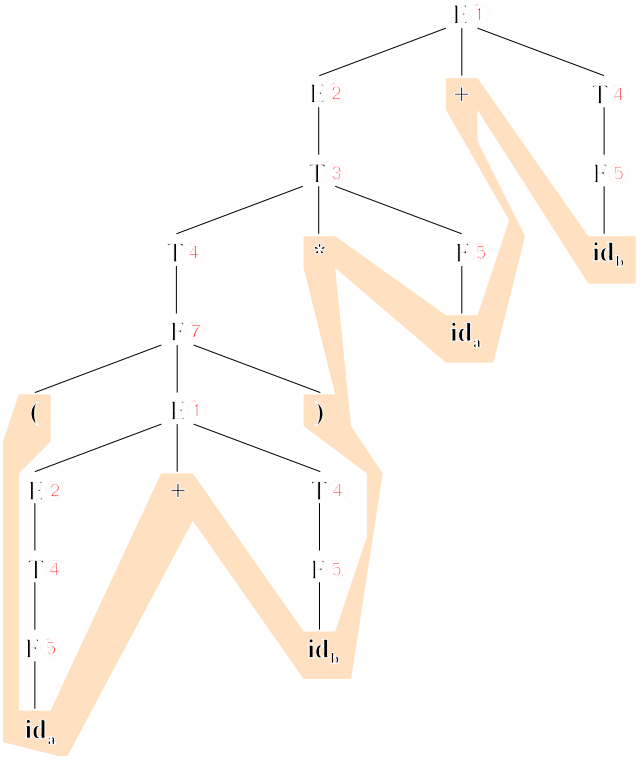


Figure 2Árbol sintáctico correspondiente al reconocimiento de la secuencia (id_a + id_b) * id_a + id_b, que se halla marcada de color.

destino en la cadena a reconocer constituye una representación del árbol sintáctico. A cada paso de derivación se añade un nuevo trozo del árbol, de forma que al comienzo del proceso se parte de un árbol que sólo tiene la raíz (el axioma inicial), al igual que las derivaciones que hemos visto parten también del axioma inicial. De esta forma, en cada derivación se selecciona un nodo hoja del árbol que se está formando y se le añaden sus hijos. En cada paso, las hojas del árbol que se va construyendo constituyen la forma sentencial por la que discurre la derivación. Por otro lado, la secuencia de derivaciones no sólo representa a un árbol sintáctico, sino que también da información de en qué orden se han ido aplicando las reglas de producción.

La construcción de este árbol conlleva la aplicación inteligente de las reglas de producción. Si la sentencia a reconocer es incorrecta, esto es, no pertenece al lenguaje generado por la gramática, será imposible obtener su árbol sintáctico. En otras ocasiones una sentencia admite más de un árbol sintáctico. Cuando esto ocurre se dice que la gramática es ambigua. Este tipo de situaciones hay que evitarlas, de suerte que,

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

o bien se cambia la gramática, o bien se aplican reglas *desambiguantes* (que veremos posteriormente). Por ejemplo, la gramática del cuadro 3.2 es ambigua puesto que ante una suma múltiple no permite distinguir entre asociatividad a la derecha o a la izquierda, esto es, ante la sentencia "aux + cont + i" que se traduce por los *tokens*: $id_{aux} + id_{cont} + id_i$ se pueden obtener los dos árboles distintos de la figura 3.3 (hemos puesto uno arriba y otro abajo, aunque ambos se formarían de la misma manera).

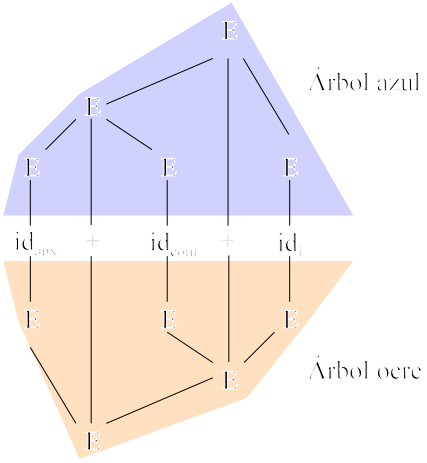


Figure 3 Ejemplo de reconocimiento de una sentencia mediante una gramática ambigua.

Los dos árboles obtenidos obedecen a dos secuencias de derivaciones distintas:

- Árbol azul:
  $E \Rightarrow E + E \Rightarrow E + id_i \Rightarrow E + E + id_i \Rightarrow E + id_{cont} + id_i \Rightarrow id_{aux} + id_{cont} + id_i$
  ↑ ↑ ↑ ↑ ↑
- Árbol ocre:
  $E \Rightarrow E + E \Rightarrow id_{aux} + E \Rightarrow id_{aux} + E + E \Rightarrow id_{aux} + id_{cont} + E \Rightarrow id_{aux} + id_{cont} + id_i$
  ↑ ↑ ↑ ↑ ↑

Como puede observarse, el quid de la cuestión está en la segunda derivación: en el árbol azul se hace una derivación a derecha, mientras que en el ocre es a izquierda; si los árboles se leen desde la cadena hacia el axioma inicial, el árbol azul da lugar a una asociatividad a la izquierda para la suma $((id_{aux} + id_{cont}) + id_i)$, mientras que el ocre produce asociatividad a derecha $(id_{aux} + (id_{cont} + id_i))$. Este hecho puede dar una idea de que la forma en que se construye el árbol sintáctico también conlleva aspectos semánticos.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

3.5 Tipos de análisis sintáctico

Según la aproximación que se tome para construir el árbol sintáctico se desprenden dos tipos o clases de analizadores:

- Descendentes: parten del axioma inicial, y van efectuando derivaciones a izquierda hasta obtener la secuencia de derivaciones que reconoce a la sentencia. Pueden ser:
    - Con retroceso.
    - Con funciones recursivas.
    - De gramáticas LL(1).
- Ascendentes: Parten de la sentencia de entrada, y van aplicando derivaciones inversas (desde el consecuente hasta el antecedente), hasta llegar al axioma inicial. Pueden ser:
    - Con retroceso.
    - De gramáticas LR(1).

Estudiaremos a continuación con detenimiento cada uno de estos métodos.

3.5.1 Análisis descendente con retroceso

Al tratarse de un método descendente, se parte del axioma inicial y se aplican en secuencia todas las posibles reglas al no terminal más a la izquierda de la forma sentencial por la que se va trabajando.

Podemos decir que todas las sentencias de un lenguaje se encuentran como nodos de un árbol que representa todas las derivaciones posibles a partir de cualquier forma sentencial, y que tiene como raíz al axioma inicial. Éste es el **árbol universal** de una gramática, de manera que sus ramas son secuencias de derivaciones y, por tanto, representan árboles sintácticos. Básicamente, podemos decir que el método de análisis descendente con retroceso pretende buscar en el árbol universal a la sentencia a reconocer; cuando lo encuentre, el camino que lo separa de la raíz nos da el árbol sintáctico. Ahora bien, es posible que la sentencia sea errónea y que no se encuentre como hoja del árbol lo que, unido a que es muy probable que el árbol sea infinito, nos lleva a la necesidad de proponer un enunciado que nos indique cuándo se debe cancelar la búsqueda porque se da por infructuosa.

Para ver un ejemplo de esto vamos a partir de la gramática del cuadro 3.3 que es equivalente a la del cuadro 3.1, aunque la que se propone ahora no es recursiva por la izquierda. Y supongamos que se desea reconocer la secuencia:

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

① E	→	T + E
②	|	T
③ T	→	F * T
④	|	F
⑤ F	→	id
⑥	|	num
⑦	|	(E)

Cuadro 3.3 Gramática no ambigua ni recursiva por la izquierda que reconoce expresiones aritméticas

(id + num) * id + num

Mediante el árbol de la figura 3.4 se pueden derivar todas las posibles sentencias reconocibles por la gramática propuesta y el algoritmo que sigue el análisis descendente con retroceso consiste hacer una búsqueda en este árbol de la rama que culmine en la sentencia a reconocer mediante un recorrido primero en profundidad.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

¿Cuando se desecha una rama que posee la forma sentencial τ? Pues, p.ej.,

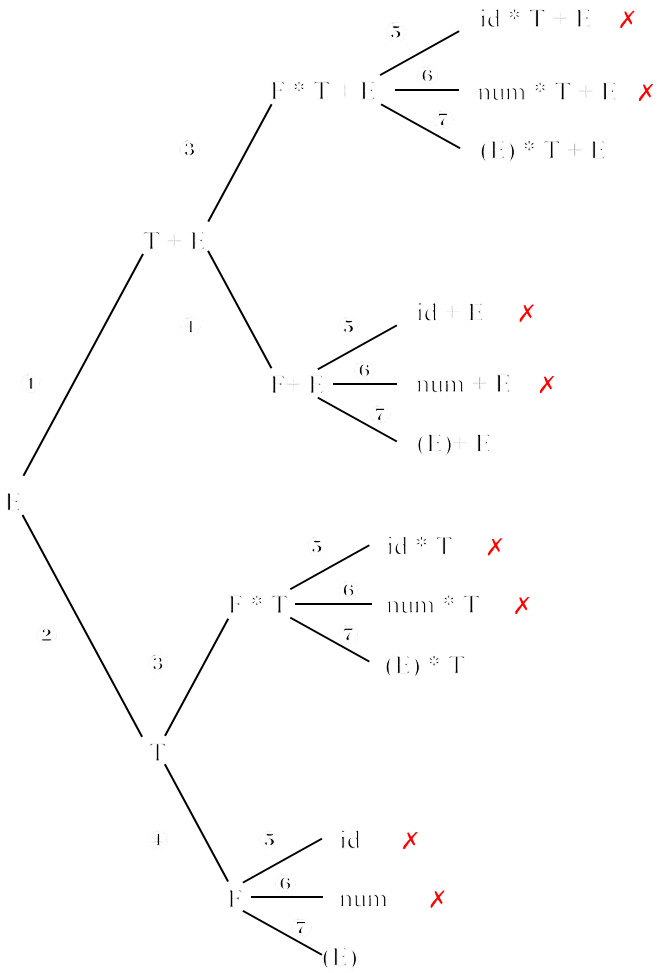


Figure 4 Árbol universal de la gramática del cuadro 3.3. Sólo se han representado las derivaciones a izquierda y los cuatro primeros niveles del árbol

cuando la secuencia de *tokens* a la izquierda del primer no terminal de τ no coincide con la cabeza de la secuencia a reconocer. Pueden establecerse otros **criterios de poda**, pero éste es el que utilizaremos en nuestro estudio.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Resumiendo, se pretende recorrer el árbol universal profundizando por cada rama hasta llegar a encontrar una forma sentencial que no puede coincidir con la sentencia que se busca, en cuyo caso se desecha la rama; o que coincide con lo buscado, momento en que se acepta la sentencia. Si por ninguna rama se puede reconocer, se rechaza la sentencia.

En el árbol universal de la figura 3.4 se han marcado de rojo las ramas por las que ya no tiene sentido continuar en base a nuestro criterio de poda. Por el resto de ramas proseguiría la búsqueda de la sentencia "(id + num) * id + num". Además, cada eje se ha numerado con el identificador de la regla que se ha aplicado.

Con el criterio de poda escogido la búsqueda en profundidad no funcionará si la gramática es recursiva a la izquierda, ya que existirán ramas infinitas en las que el número de terminales a la izquierda de una forma sentencial no aumenta y nuestra búsqueda se meterá en una recursión infinita. En estos casos es necesario incluir nuevos criterios de poda.

Los analizadores sintácticos con retroceso no suelen utilizarse en la práctica. Esto se debe a que el retroceso es sumamente ineficiente y existen otros tipos de análisis más potentes basados en mecanismos diferentes, como veremos más adelante. Incluso para el lenguaje natural el retroceso no es nada eficiente, y se prefieren otros métodos.

A continuación se presenta el algoritmo de análisis descendente con retroceso:

```
Precondición: formaSentencial ≡ μ A δ, con μ ∈ T*, A ∈ N, y δ ∈ (N ∪ T)*
AnálisisDescendenteConRetroceso(formaSentencial, entrada)
    Si μ ≠ parteIzquierda(entrada) Retornar Fin Si
    Para cada  pi ∈ {pi ∈ P / pi ≡ A → αi}
        formaSentencial´≡ μ αi δ ≡ μ´ A´ δ´,        con μ´ ∈ T*, A´ ∈ N,
                                                    y δ´ ∈ (N ∪ T)*
        Si μ´ ≡ entrada
            ¡ Sentencia reconocida !
            Retornar
        Fin Si
        AnálisisDescendenteConRetroceso(formaSentencial´, entrada)
        Si ¡ Sentencia reconocida ! Retornar Fin Si
    Fin For
Fin AnálisisDescendenteConRetroceso
```

Y la llamada principal quedaría:

```
AnálisisDescendenteConRetroceso(axiomaInicial, cadenaAReconocer)
Si NOT ¡ Sentencia reconocida !
        ¡¡ Sentencia no reconocida !!
Fin Si
```

La tabla siguiente muestra la ejecución de este algoritmo para reconocer o rechazar la sentencia "(id + num) * id + num". Se aplican siempre derivaciones a izquierda, y la columna "Pila de reglas utilizadas" representa la rama del árbol

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

universal sobre la que se está trabajando. Se han representado en rojo los retrocesos. Nótese como la tabla es mucho más extensa de lo que aquí se ha representado, lo que da una estimación de la ineficiencia del algoritmo.

Forma sentencial	Pila de reglas utilizadas
E	1
T + E	1-3
F * T +E	1-3-5
id * T + E	1-3
F * T +E	1-3-6
num * T + E	1-3
F * T +E	1-3-7
(E) * T +E	1-3-7-1
(T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-5
(id * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-6
(num * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1-3-7
((E) * T + E) * T + E	1-3-7-1-3
(F * T + E) * T + E	1-3-7-1
(T + E) * T + E	1-3-7-1-4
(F + E) * T + E	1-3-7-1-4-5
(id + E) * T + E	1-3-7-1-4-5-1
(id + T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1-3-5
(id + id * T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1-3-6
(id + num * T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1-3-7
(id + (E) * T + E) * T + E	1-3-7-1-4-5-1-3
(id + F * T + E) * T + E	1-3-7-1-4-5-1
(id + T + E) * T + E	1-3-7-1-4-5-1-4
(id + F + E) * T + E	
....	
(id + E) * T + E	1-3-7-1-4-5-2
(id + T) * T + E	
...	

3.5.2 Análisis descendente con funciones recursivas

Una gramática de contexto libre constituye un mecanismo para expresar un lenguaje formal. Sin embargo no es el único, ya que la notación BNF y los diagramas de sintaxis tienen la misma potencia.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Partiendo de que la notación BNF también permite expresar un lenguaje formal, a continuación se dará una definición de diagrama de sintaxis y se demostrará que éstos tienen, como mínimo, la misma potencia que la mencionada notación BNF. A partir de este punto, seremos capaces de construir *ad hoc* un analizador sintáctico mediante cualquier lenguaje de programación que permita la recursión.

3.5.2.1 Diagramas de sintaxis

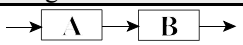
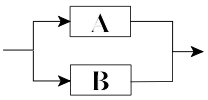
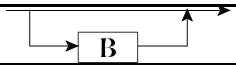
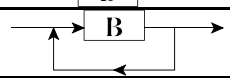
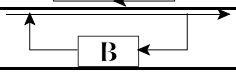
Un diagrama de sintaxis (también llamados diagramas de Conway) es un grafo dirigido donde los elementos no terminales aparecen como rectángulos, y los terminales como círculos o elipses.

Todo diagrama de sintaxis posee un origen y un destino, que no se suelen representar explícitamente, sino que se asume que el origen se encuentra a la izquierda del diagrama y el destino a la derecha.

Cada arco con origen en α y destino en β representa que el símbolo α puede ir seguido del β (pudiendo ser α y β tanto terminales como no terminales). De esta forma todos los posibles caminos desde el inicio del grafo hasta el final, representan formas sentenciales válidas.

3.5.2.2 Potencia de los diagramas de sintaxis

Demostraremos que los diagramas de sintaxis permiten representar las mismas gramáticas que la notación BNF, por inducción sobre las operaciones básicas de BNF:

Operación	BNF	Diagrama de sintaxis
Yuxtaposición	AB	
Opción	A | B	
	ε | B	
Repetición	1 o más veces {B}	
	0 o más veces [B]	

Siguiendo la «piedra de Rosetta» que supone la tabla anterior, es posible convertir cualquier expresión BNF en su correspondiente diagrama de sintaxis, ya que A y B pueden ser, a su vez, expresiones o diagramas complejos.

Por otro lado, quien haya trabajado con profusión los diagramas de sintaxis habrá observado que no siempre resulta fácil convertir un diagrama de sintaxis cualquiera a su correspondiente en notación BNF.

3.5.2.3 Correspondencia con flujos de ejecución

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Ahora vamos a establecer una correpondencia entre el flujo de ejecución de un programa y el camino que se puede seguir en uno o varios diagramas de sintaxis para reconocer una sentencia válida. Por ejemplo, la yuxtaposición quedaría como ilustra la figura 3.10.

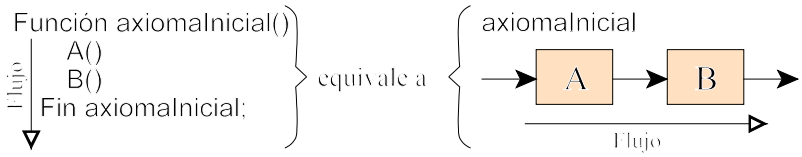


Figure 10 Equivalencia entre el diagrama de sintaxis de yuxtaposición y una función que sigue el mismo flujo. La función se llama igual que el diagrama.

Es importante el ejercicio de programación consistente en seguir la evolución detallada de las llamadas de los procedimientos entre sí. Antes de comenzar con un ejemplo completo conviene reflexionar sobre cómo actuar en caso de que nos encontremos con un símbolo terminal en el diagrama de sintaxis. En tales situaciones habrá que controlar que el *token* que nos devuelve el analizador léxico coincide con el que espera el diagrama de sintaxis. Por ejemplo, el diagrama de la figura 3.11 se traduciría por el bloque de código:

```
Función prog ( )
        sent ( );
```

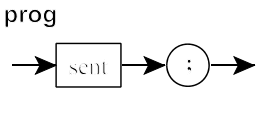


Figure 11 Un diagrama de sintaxis para prog

```
        Si el siguiente token es ';' entonces
                consumir el token
        si no
                ¡Error sintáctico!
        Fin si
Fin prog
```

donde consumir el token hace referencia a que se le pide el siguiente *token* al analizador léxico. Para el resto de ejemplos supondremos que el analizador léxico suministra una función denominada get_token() que almacena el siguiente *token* en la variable global token. Siguiendo este criterio, si partiéramos del diagrama de la figura 3.12 el código generado sería:

```
Función prog2 ( );
        get_token()
        sent ( );
        Si token == ';' entonces
                get_token()
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
            si no
                    ¡Error sintáctico!
            Fin si
            Si token == FIN entonces
                    get_token()
            si no
                    ¡Error sintáctico!
            Fin si
Fin Prog2
```

Es importante darse cuenta de que para que el analizador sintáctico pueda comenzar el reconocimiento, la variable token debe tener ya un valor pre-cargado; a esta variable se la llama *token* de prebúsqueda (*lookahead*), y es de fundamental importancia, puesto que es la que nos permitirá tomar la decisión de por dónde debe continuar el flujo en el diagrama de sintaxis, como veremos más adelante. También es importante apreciar que lo que aquí se ha denotado genéricamente por ¡Error sintáctico! consiste realmente en un mensaje indicativo del número de línea en que se ha encontrado el error, así como su naturaleza, p.ej.: «Se esperaba una ‘,’ y se ha encontrado token»

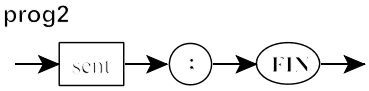


Figure 12 Un nuevo diagrama de sintaxis para prog: prog2

3.5.2.4 Ejemplo completo

En este apartado vamos a construir las funciones recursivas para una serie de diagramas que representan un lenguaje consistente en expresiones lógico/aritméticas finalizada cada una de ellas en punto y coma. El axioma inicial viene dado por el no terminal secuencia, quien también debe encargarse de hacer el primer get_token(), así como de controlar que el último *pseudotoken* enviado por el analizador léxico sea **EOF** (*End Of File*).

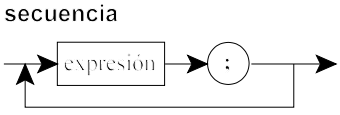


Figure 13 Diagrama de sintaxis de secuencia

```
Función secuencia ( ) { // Figura 3.13
        get_token ( );
        do {
            expresión ( );
            while (token != PUNTOYCOMA) {
                !Error en expresión¡
                get_token ( );
            };
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
            get_token( );
        } while (token != EOF);
};
```

En este caso se considera al ';' (PUNTOYCOMA) como un *token* de seguridad, lo que permite hacer una recuperación de errores mediante el método *panic mode* (ver punto 3.3.1).

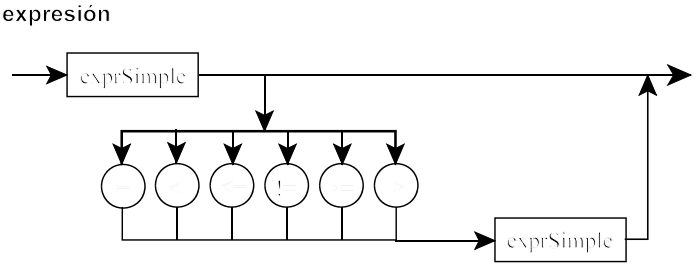


Figure 14 Diagrama de sintaxis de expresión

```
Función expresión() { // Figura 3.14
        exprSimple();
        if ((token == IGUAL)|| (token == ME)|| (token == MEI)||
           (token == DIST)|| (token == MAI)|| (token == MA)) {
                get_token();
                exprSimple();

        }
}
```

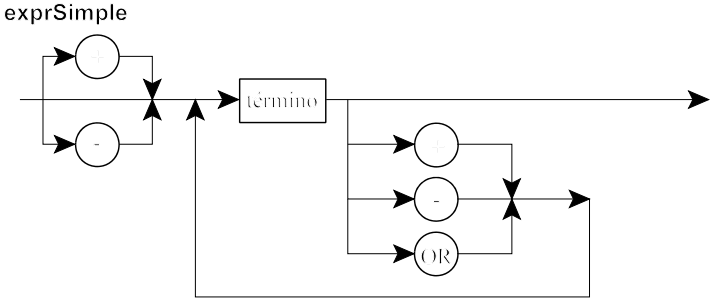


Figure 15 Diagrama de sintaxis de exprSimple

```
Función exprSimple ( ) { // Figura 3.15
        if ((token == MAS) || (token == MENOS)) {
          get_token( );
        }
        término ( );
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
            while ((token == MAS) || (token == MENOS) || (token == OR)) {
              get_token( );
              término ( );
            }
}
```

término

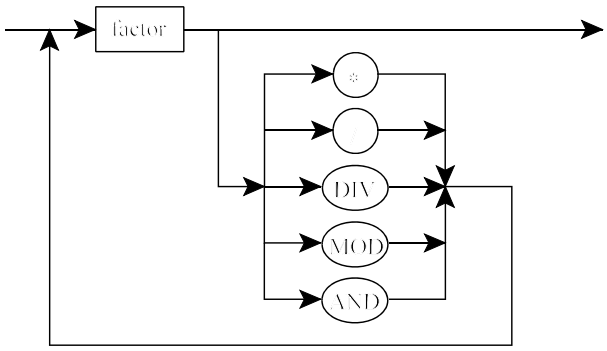


Figure 16 Diagrama de sintaxis de término

```
Función término() { // Figura 3.16
        factor();
        while ((token == POR) || (token == DIV) || (token == DIV_ENT) ||
                (token == MOD) || (token == AND) {
                    get_token();
                    factor();
        }
}
```

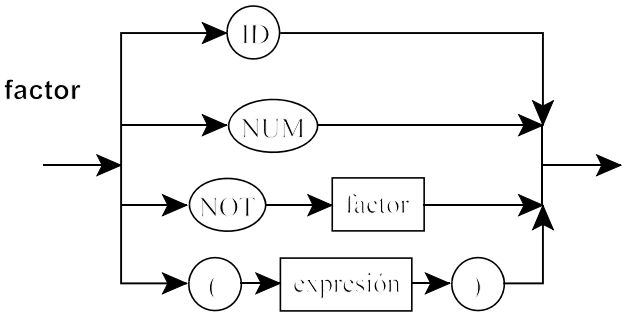


Figure 17 Diagrama de sintaxis de factor

```
Función factor ( ){ // Figura 3.17
        switch (token) {
           case ID : get_token(); break;
```


Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

```
                case NUM : get_token(); break;
                case NOT : get_token(); factor(); break;
                case AB_PARID :     get_token();
                                    expresión();
                                    if (token != CE_PAR) {
                                       Error: Paréntesis de cierre
                                    } else get_token();
                                    break;
                default : Error: Expresión no válida.
            }
}
```

Este ejemplo muestra cómo resulta relativamente fácil obtener funciones recursivas que simulen en ejecución el camino seguido por un diagrama de sintaxis para reconocer una cadena.

3.5.2.5 Conclusiones sobre el análisis descendente con funciones recursivas

No todo diagrama de sintaxis es susceptible de ser convertido tan fácilmente en función recursiva; concretamente, en aquellos nudos del diagrama en los que el flujo puede seguir por varios caminos, la decisión sobre cuál seguir se ha hecho en base al siguiente *token* de la entrada, de manera que dicho *token* actúa como discriminante. El diagrama de la figura 3.18.a ilustra un ejemplo en el cual necesitamos más de un *token* (concretamente dos) para tomar correctamente la decisión de qué camino seguir. Por regla general, estos diagramas pueden reconstruirse de manera que se adapten bien a

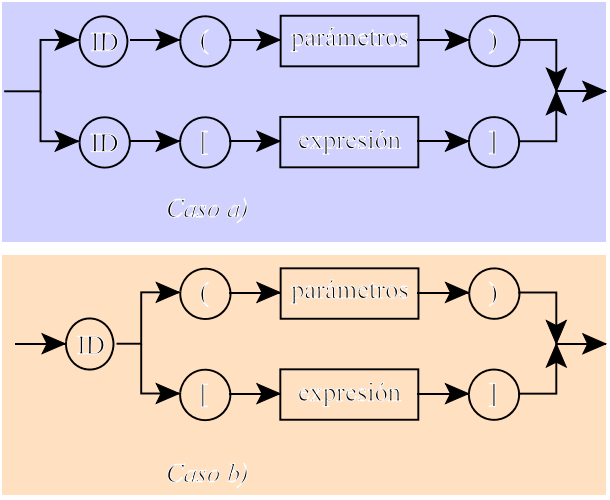


Figure 18 Diagramas de sintaxis equivalentes. El caso a) tiene dos alternativas que comienzan por el mismo *token*. El caso b) ha eliminado este problema

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

nuestro mecanismo de construcción de funciones, tal y como ilustra la figura 3.18.b.

El metacompilador JavaCC utiliza la notación BNF para expresar una gramática, y construye una función recursiva por cada no terminal de la gramática incluyendo en ella toda la lógica interna de reconocimiento. En los casos de conflicto, como el ilustrado anteriormente, el desarrollador del compilador puede darle a JavaCC una indicación para que utilice un mayor número de *tokens* de pre-búsqueda.

3.5.3 Análisis descendente de gramáticas LL(1)

Este tipo de análisis se basa en un autómata de reconocimiento en forma de tabla, denominada tabla de chequeo de sintaxis. Dicha tabla posee como eje de ordenadas a los no terminales de la gramática, y como abcisas a los terminales (incluido en *pseudotoken* EOF). El contenido de cada casilla interior (que se corresponde con la columna de un terminal y la fila de un no terminal) contiene, o bien una regla de producción, o bien está vacía, lo que se interpreta como un rechazo de la cadena a reconocer.

En general, podemos decir que una gramática LL(1) es aquélla en la que es suficiente con examinar sólo un símbolo a la entrada, para saber qué regla aplicar, partiendo del axioma inicial y realizando derivaciones a izquierda. Toda gramática reconocible mediante el método de los diagramas de sintaxis siguiendo el procedimiento visto en el punto anterior es LL(1).

El método consiste en seguir un algoritmo partiendo de:

- La cadena a reconocer, junto con un apuntador, que nos indica cual es el *token* de pre-búsqueda actual; denotaremos por **a** dicho *token*.
- Una pila de símbolos (terminales y no terminales); denotaremos por **X** la cima de esta pila.
- Una tabla de chequeo de sintaxis asociada de forma unívoca a una gramática. En el presente texto no se trata la forma en que dicha tabla se obtiene a partir de la gramática. Denotaremos por **M** a dicha tabla, que tendrá una fila por cada no terminal de la gramática de partida, y una columna por cada terminal incluido el EOF: $M[N \times T \cup \{\$\}]$.

Como siempre, la cadena de entrada acabará en EOF que, a partir de ahora, denotaremos por el símbolo '$'. El mencionado algoritmo consiste en consultar reiteradamente la tabla **M** hasta aceptar o rechazar la sentencia. Partiendo de una pila que posee el $ en su base y el axioma inicial S de la gramática encima, cada paso de consulta consiste en seguir uno de los puntos siguientes (son excluyentes):

1.- Si $X = a = \$$ entonces ACEPTAR.

2.- Si $X = a \neq \$$ entonces
- se quita X de la pila
- y se avanza el apuntador.

3.- Si $X \in T$ y $X \neq a$ entonces RECHAZAR.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

4.- Si X ∈ N entonces consultamos la entrada M[X,a] de la tabla, y :

- Si M[X,a] es vacia : RECHAZAR.
- Si M [X,a] no es vacia, se quita a X de la pila y se inserta el consecuente en orden inverso. Ejemplo: Si M[X,a] = {X → UVY}, se quita a X de la pila, y se meten UVY en orden inverso, como muestra la figura 3.19.

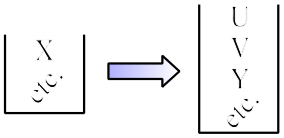


Figure 19 Aplicación de regla en el método LL(1)

Al igual que sucedía en el análisis descendente con funciones recursivas, una vez aplicada una regla, ésta ya no será desaplicada por ningún tipo de retroceso.

Uno de los principales inconvenientes de este tipo de análisis es que el número de gramáticas LL(1) es relativamente reducido; sin embargo, cuando una gramática no es LL(1), suele ser posible traducirla para obtener una equivalente que sí sea LL(1), tras un adecuado estudio. Por ejemplo, la siguiente gramática no es LL(1):

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

pero puede ser factorizada, lo que la convierte en:

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

que es equivalente y LL(1), siendo su tabla M:

T / N	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → F T'			T → F T'		
T'		T' → ε	T' → * F T'		T' → ε	T' → ε
F	F → id			F → (E)		

Evidentemente, esta tabla pertenece a la gramática y puede emplearse para rechazar o reconocer cualesquiera sentencias, esto es, la tabla no se construye para cada sentencia a reconocer, sino que está asociada exclusivamente a la gramática y viene a

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

ser una representación de su autómata finito con pila.

La secuencia siguiente muestra la ejecución de este algoritmo para reconocer o rechazar la sentencia "id * (id + id) $". La columna "Acción" indica qué punto del algoritmo es el que se aplica, mientras que la columna "Cadena de Entrada" indica, a la izquierda, qué *tokens* se han consumido, a la derecha lo que quedan por consumir y, subrayado, el *token* de pre-búsqueda actual. Puede observarse cómo la ejecución de este método es mucho más eficiente que el caso con retroceso, e igual de eficiente que el análisis con funciones recursivas.

Pila de símbolos	Cadena de Entrada	Acción
$ E	id * (id + id) $	4.- M[E, id] = E → T E'
$ E' T	id * (id + id) $	4.- M[T, id] = T → F T'
$ E' T' F	id * (id + id) $	4.- M[F, id] = F → id
$ E' T' id	id * (id + id) $	2.- id = id
$ E' T'	id * (id + id) $	4.- M[T', *] = T' → * F T'
$ E' T' F *	id * (id + id) $	2.- * = *
$ E' T' F	id * (id + id) $	4.- M[F, (] = F → (E)
$ E' T') E (	id * (id + id) $	2.- (= (
$ E' T') E	id * (id + id) $	4.- M[E, id] = E → T E'
$ E' T') E' T	id * (id + id) $	4.- M[T, id] = T → F T'
$ E' T') E' T' F	id * (id + id) $	4.- M[F, id] = F →id
$ E' T') E' T' id	id * (id + id) $	2.- id = id
$ E' T') E' T'	id * (id + id) $	4.- M[T',+] = T' → ε
$ E' T') E'	id * (id + id) $	4.- M[E',+] = E' →+ T E'
$ E' T') E' T +	id * (id + id) $	2.- + = +
$ E' T') E' T	id * (id + id) $	4.- M[T, id] = T →F T'
$ E' T') E' T' F	id * (id + id) $	4.- M[F, id] = F →id
$ E' T') E' T' id	id * (id + id) $	2.- id = id
$ E' T') E' T'	id * (id + id) $	4.- M[T',)] = T' → ε
$ E' T') E'	id * (id + id) $	4.- M[E',)] = E' → ε
$ E' T')	id * (id + id) $	2.-) =)
$ E' T'	id * (id + id) $	4.- M[T', $] = T' → ε
$ E'	id * (id + id) $	4.- M[E', $] = E' → ε
$	id * (id + id) $	1.- $ = $ ACEPTAR

La secuencia de reglas aplicadas (secuencia de pasos de tipo 4) proporciona la secuencia de derivaciones a izquierda que representa el árbol sintáctico que reconoce la sentencia.

Este método tiene una ventaja sobre el análisis con funciones recursivas, y es que la construcción de la tabla de chequeo de sintaxis puede automatizarse, por lo que una modificación en la gramática de entrada no supone un problema grande en lo que a reconstrucción de la tabla se refiere. No sucede lo mismo con las funciones recursivas, ya que estas tienen que ser reconstruidas a mano con la consecuente pérdida de tiempo y propensión a errores.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

3.5.4 Generalidades del análisis ascendente

Antes de particularizar en los distintos tipos de análisis ascendentes, resulta conveniente estudiar los conceptos y metodología comunes a todos ellos. El objetivo de un análisis ascendente consiste en construir el árbol sintáctico desde abajo hacia arriba, esto es, desde los *tokens* hacia el axioma inicial, lo cual disminuye el número de reglas mal aplicadas con respecto al caso descendente (si hablamos del caso con retroceso) o amplia el número de gramáticas susceptibles de ser analizadas (si hablamos del caso LL(1)).

Tanto si hay retroceso como si no, en un momento dado, la cadena de entrada estará dividida en dos partes, denominadas α y β:

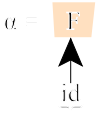
β: representa el trozo de la cadena de entrada (secuencia de *tokens*) por consumir: $\beta \in T^*$. Coincidirá siempre con algún trozo de la parte derecha de la cadena de entrada. Como puede suponerse, inicialmente β coincide con la cadena a reconocer al completo (incluido el EOF del final).

α: coincidirá siempre con el resto de la cadena de entrada, trozo al que se habrán aplicado algunas reglas de producción en sentido inverso: $\alpha \in (N \cup T)^*$.

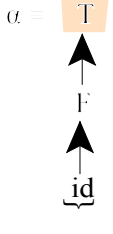
Por ejemplo, si quisiéramos reconocer "id + id + id", partiendo de la gramática del cuadro 3.3 se comenzaría a construir el árbol sintáctico a partir del árbol vacío (α = ε) y con toda la cadena de entrada por consumir (β = id + id + id):

$\underbrace{\varepsilon}_{\alpha}$ $\underbrace{\text{id} + \text{id} + \text{id}}_{\beta}$, y tras consumir el primer *token*:

$\underbrace{\text{id}}_{\alpha}$ $\underbrace{+\,\text{id} + \text{id}}_{\beta}$, y ahora podemos aplicar la regla ⑤ hacia atrás, con lo que α es F:

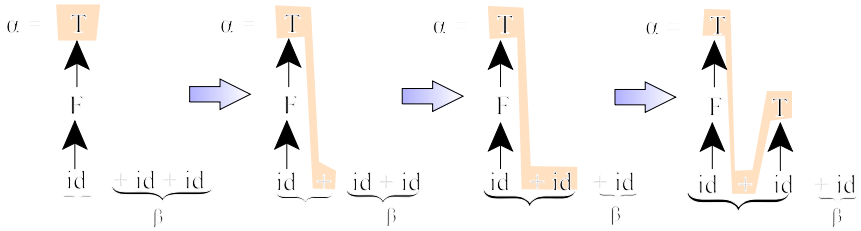


, a lo que ahora se puede aplicar la regla ④, produciendo:



. Así, poco a poco se va construyendo el árbol:

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike



Como puede verse, α representa al árbol sintáctico visto desde arriba conforme se va construyendo ascendentemente: α es una forma sentencial ascendente.

3.5.4.1 Operaciones en un analizador ascendente

A medida que un analizador sintáctico va construyendo el árbol, se enfrenta a una configuración distinta (se denomina configuración al par α-β) y debe tomar una decisión sobre el siguiente paso u operación a realizar. Básicamente se dispone de cuatro operaciones diferentes, y cada tipo de analizador ascendente se distingue de los demás en base a la inteligencia sobre cuándo aplicar cada una de dichas operaciones. Cualquier mecanismo de análisis ascendente consiste en partir de una configuración inicial e ir aplicando operaciones, cada una de las cuales permite pasar de una configuración origen a otra destino. El proceso finalizará cuando la configuración destino llegue a ser tal que α represente al árbol sintáctico completo y en β se hayan consumido todos los *tokens*. Las operaciones disponibles son las siguientes:

1.- ACEPTAR: se acepta la cadena: β ≡ EOF y α ≡ S (axioma inicial).

2.- RECHAZAR: la cadena de entrada no es válida.

3.- **Reducir**: consiste en aplicar una regla de producción hacia atrás a algunos elementos situados en el extremo derecho de α. Por ejemplo, si tenemos la configuración:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ X_{p+1}\ \ldots\ X_m]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n]$$

(recordemos que $X_i \in (N \cup T)^*$ y $a_i \in T$), y existe una regla de la forma

$$A_k \rightarrow X_{p+1}\ \ldots\ X_m$$

entonces una reducción por dicha regla consiste en pasar a la configuración:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ A_k]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n].$$

Resulta factible reducir por reglas épsilon, esto es, si partimos de la configuración:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ X_{p+1}\ \ldots\ X_m]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n]$$

puede reducirse por una regla de la forma:

$$A_k \rightarrow \varepsilon$$

obteniéndose:

$$\alpha \equiv [X_1\ X_2\ \ldots\ X_p\ X_{p+1}\ \ldots\ X_m\ A_k]\text{-}\beta \equiv [a_{m+1}\ \ldots\ a_n]$$

4.- **Desplazar**: consiste únicamente en quitar el terminal más a la izquierda de β y ponerlo a la derecha de α. Por ejemplo, si tenemos la configuración:

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

$$\alpha \equiv [X_1\ X_2\ ...\ X_p\ X_{p+1}\ ...\ X_m]\text{-}\beta \equiv [a_{m+1}\ a_{m+2}\ ...\ a_n]$$

un desplazamiento pasaría a:

$$\alpha \equiv [X_1\ X_2\ ...\ X_p\ X_{p+1}\ ...\ X_m\ a_{m+1}]\text{-}\beta \equiv [a_{m+2}\ ...\ a_n]$$

Resumiendo, mediante reducciones y desplazamientos, tenemos que llegar a aceptar o rechazar la cadena de entrada. Antes de hacer los desplazamientos tenemos que hacerles todas las reducciones posibles a α, puesto que éstas se hacen a la parte derecha de α, y no por enmedio. Cuando α es el axioma inicial y β es la tira nula (sólo contiene EOF), se acepta la cadena de entrada. Cuando β no es la tira nula o α no es el axioma inicial y no se puede aplicar ninguna regla, entonces se rechaza la cadena de entrada.

3.5.5 Análisis ascendente con retroceso

Al igual que ocurría con el caso descendente, este tipo de análisis intenta probar todas las posibles operaciones (reducciones y desplazamientos) mediante un método de fuerza bruta, hasta llegar al árbol sintáctico, o bien agotar todas las opciones, en cuyo caso la cadena se rechaza. El algoritmo es el siguiente:

```
Precondición: α ∈ (N ∪ T)* y β ∈ T*
AnálisisAscendenteConRetroceso(α, β)
    Para cada p_i ∈ P hacer
        Si consecuente(p_i) ≡ cola(α)
            α'-β' = reducir α-β por la regla p_i
            Si α ≡ S AND β≡ ε
                ¡ Sentencia reconocida ! (ACEPTAR)
            si no
                AnálisisAscendenteConRetroceso(α', β')
            Fin si
        Fin si
    Fin para
    Si β ≠ ε
        α'-β' = desplazar α-β
        AnálisisAscendenteConRetroceso(α', β')
    Fin si
Fin AnálisisAscendenteConRetroceso
```

Y la llamada principal quedaría:

```
AnálisisAscendenteConRetroceso(ε, cadenaAReconocer)
Si NOT ¡ Sentencia reconocida !
        ¡¡ Sentencia no reconocida !! (RECHAZAR)
Fin Si
```

Como el lector habrá podido apreciar, este algoritmo incurre en recursión sin fin ante gramáticas que posean reglas épsilon (esto es, que permitan la cadena vacía como parte del lenguaje a reconocer). Esto se debe a que una regla épsilon siempre puede aplicarse hacia atrás, dando lugar a una rama infinita hacia arriba. P.ej., sea cual sea la cadena a reconocer, si existe una regla épsilon F → ε se obtendrá mediante este

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

algoritmo algo parecido a la figura 3.24

Retomemos a continuación la gramática del cuadro 3.1 y veamos el proceso que se sigue para reconocer la cadena "id * id":

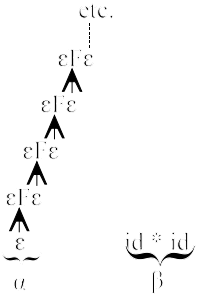


Figure 24Nótese como la cola de α puede ser ε, Fε, εFε, etc., lo que hace que la regla F → ε pueda aplicarse indefinidamente.

Pila de reglas utilizadas	α	β	Acción
--	ε	id * id	Desplazar
--	id	* id	Reducir por F → id
5	F	* id	Reducir por T → F
5-4	T	* id	Reducir por E → T
5-4-2	E	* id	Desplazar
5-4-2	E *	id	Desplazar
5-4-2	E * id	ε	Reducir por F → id
5-4-2-5	E * F	ε	Reducir por T → F
5-4-2-5-4	E * T	ε	Reducir por E → T
5-4-2-5-4-2	E * E	ε	Retroceso (pues no hay nada por desplazar)
5-4-2-5-4	E * T	ε	Retroceso (pues no hay nada por desplazar)
5-4-2-5	E * F	ε	Retroceso (pues no hay nada por desplazar)
5-4-2	E * id	ε	Retroceso (pues no hay nada por desplazar)
5-4-2	E *	id	Retroceso (pues ya se desplazó)
5-4-2	E	* id	Retroceso (pues ya se desplazó)
5-4	T	* id	Desplazar
5-4	T *	id	Desplazar
5-4	T * id	ε	Reducir por F → id
5-4-5	T * F	ε	Reducir por T → T * F
5-4-5-3	T	ε	Reducir por E → T
5-4-5-3-2	E	ε	Aceptar

Este método es más eficiente que el descendente con retroceso puesto que consume los *tokens* a mayor velocidad y, por tanto, trabaja con mayor cantidad de información a la hora de tomar cada decisión. No obstante resulta inviable para aplicaciones prácticas pues su ineficiencia sigue siendo inadmisible.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

3.5.6 Análisis ascendente de gramáticas LR(1)

En este epígrafe se introducirá una técnica eficiente de análisis sintáctico ascendente que se puede utilizar para procesar una amplia clase de gramáticas de contexto libre. La técnica se denomina análisis sintáctico LR(k). La abreviatura LR obedece a que la cadena de entrada es examinada de izquierda a derecha (en inglés, *Left-to-right*), mientras que la "R" indica que el proceso proporciona el árbol sintáctico mediante la secuencia de derivaciones a derecha (en inglés, *Rightmost derivation*) en orden inverso. Por último, la "k" hace referencia al número de *tokens* de pre-búsqueda utilizados para tomar las decisiones sobre si reducir o desplazar. Cuando se omite, se asume que k, es 1.

El análisis LR es atractivo por varias razones.

- Pueden reconocer la inmensa mayoría de los lenguajes de programación que puedan ser generados mediante gramáticas de contexto-libre.
- El método de funcionamiento de estos analizadores posee la ventaja de localizar un error sintáctico casi en el mismo instante en que se produce con lo que se adquiere una gran eficiencia de tiempo de compilación frente a procedimientos menos adecuados como puedan ser los de retroceso. Además, los mensajes de error indican con precisión la fuente del error.

El principal inconveniente del método es que supone demasiado trabajo construir manualmente un analizador sintáctico LR para una gramática de un lenguaje de programación típico, siendo necesario utilizar una herramienta especializada para ello: un generador automático de analizadores sintácticos LR. Por fortuna, existen disponibles varios de estos generadores. Más adelante estudiaremos el diseño y uso de algunos de ellos: los metacompiladores Yacc, Cup y JavaCC. Con Yacc o Cup se puede escribir una gramática de contexto libre generándose automáticamente su analizador sintáctico. Si la gramática contiene ambigüedades u otras construcciones difíciles de analizar en un examen de izquierda a derecha de la entrada, el metacompilador puede localizar las reglas de producción problemáticas e informar de ellas al diseñador.

Esta técnica, al igual que la del LL(1), basa su funcionamiento en la existencia de una tabla especial asociada de forma única a una gramática. Existen varias técnicas para construir dicha tabla, y cada una de ellas produce un "sucedáneo" del método principal. La mayoría de autores se centra en tres métodos principales. El primero de ellos, llamado LR sencillo (*SLR*, en inglés) es el más fácil de aplicar, pero el menos poderoso de los tres ya que puede que no consiga producir una tabla de análisis sintáctico para algunas gramáticas que otros métodos sí consiguen. El segundo método, llamado LR canónico, es el más potente pero el más costoso. El tercer método, llamado LR con examen por anticipado (*Look Ahead LR*, en inglés), está entre los otros dos en cuanto a potencia y coste. El método LALR funciona con las gramáticas de la mayoría de los lenguajes de programación y, con muy poco esfuerzo, se puede implantar de forma eficiente. Básicamente, un método es más potente que otro en función del número de gramáticas a que puede aplicarse. Mientras más amplio sea el espectro de gramáticas admitidas más complejo se vuelve el método.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Funcionalmente hablando, un analizador LR consta de dos partes bien diferenciadas: a) un programa de proceso y b) una tabla de análisis. El programa de proceso posee, como se verá seguidamente, un funcionamiento muy simple y permanece invariable de analizador a analizador. Según sea la gramática a procesar deberá variarse el contenido de la tabla de análisis que es la que identifica plenamente al analizador.

La figura 3.25 muestra un esquema sinóptico de la estructura general de una analizador LR. Como puede apreciarse en ella, el analizador procesa una cadena de entrada finalizada con el símbolo $ que representa el delimitador EOF. Esta cadena se lee de izquierda a derecha, y el reconocimiento de un solo símbolo permite tomar las decisiones oportunas (LR(1)).

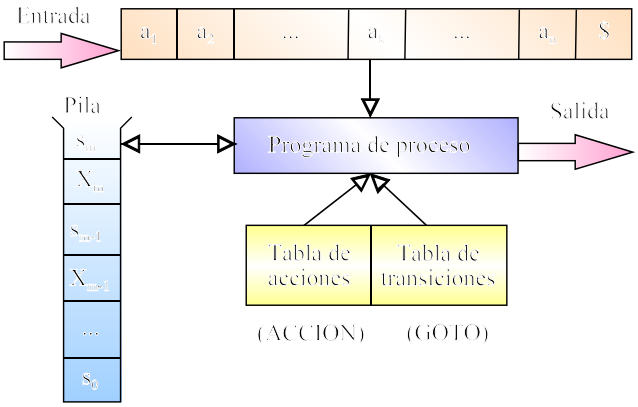


Figure 25 Esquema de funcionamiento de un analizador LR cualquiera

Además, el algoritmo hace uso de una pila que tiene la forma:

$$s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m$$

donde el símbolo s_m se encuentra en la cima tal y como se muestra en la figura 3.25. Cada uno de los X_i es un símbolo de la gramática ($X_i \in (N \cup T)^*$) y los s_i representan distintos estados del autómata asociado a la gramática; al conjunto de estados lo denominaremos E: $s_i \in E$. Cada estado s_i tiene asociado un símbolo X_i, excepto el s_0 que representa al estado inicial y que no tiene asociado símbolo ninguno. Los estados se utilizan para representar toda la información contenida en la pila y situada antes del propio estado. Consultando el estado en cabeza de la pila y el siguiente *token* a la entrada se decide qué reducción ha de efectuarse o bien si hay que desplazar.

Por regla general una tabla de análisis para un reconocedor LR consta de dos partes claramente diferenciadas entre sí que representan dos tareas distintas, la tarea de

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

transitar a otro estado (GOTO) y la tarea de qué acción realizar (ACCION). La tabla GOTO es de la forma $E \times (N \cup T \cup \{\$\})$ y contiene estado de E, mientras que la tabla ACCION es de la forma $E \times (N \cup T)$ y contiene una de las cuatro acciones que vimos en el apartado 3.5.4.

Suponiendo que en un momento dado el estado que hay en la cima de la pila es s_m y el *token* actual es a_i, el funcionamiento del analizador LR consiste en aplicar reiteradamente los siguientes pasos, hasta aceptar o rechazar la cadena de entrada:

1.- Consultar la entrada (s_m, a_i) en la tabla de ACCION. El contenido de la casilla puede ser:

$$\text{ACCION}(s_m, a_i) \equiv \begin{cases} \text{Aceptar} \\ \text{Rechazar} \\ \text{Reducir por } A \rightarrow \tau \\ \text{Desplazar} \end{cases}$$

Si se acepta o rechaza, se da por finalizado el análisis, si se desplaza se mete a_i en la pila, y si se reduce, entonces la cima de pila coincide con el consecuente τ (amén de los estados), y se sustituye τ por A.

2.- Una vez hecho el proceso anterior, en la cima de la pila hay un símbolo y un estado, por este orden, lo que nos dará una entrada en la tabla de GOTO. El contenido de dicha entrada se coloca en la cima de la pila.

Formalmente se define una **configuración de un analizador LR** como un par de la forma:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m \ , \ a_i a_{i+1} \ldots a_n \$)$$

es decir, el primer componente es el contenido actual de la pila, y el segundo la subcadena de entrada que resta por reconocer, siendo a_i el *token* de pre-búsqueda. Partiremos de esta configuración general para el estudio que prosigue.

Así, formalmente, cada ciclo del algoritmo LR se describe como:

1.1.- Si $\text{ACCION}(s_m, a_i)$ = Desplazar, entonces se introduce en la pila el símbolo a_i, produciendo:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i \ , \ a_{i+1} \ldots a_n \$)$$

A continuación se consulta la entrada (s_m, a_i) de la tabla GOTO: $\text{GOTO}(s_m, a_i) = s_{m+1}$, produciéndose finalmente:

$$(s_0 X_1 s_1 X2 s_2 \ldots X_m s_m a_i \ s_{m+1} \ , \ a_{i+1} \ldots a_n \$)$$

pasando s_{m+1} a estar situado en cabeza de la pila y a_{i+1} el siguiente símbolo a explorar en la cinta de entrada.

1.2.- Si $\text{ACCION}(s_m, a_i)$ = Reducir por $A \rightarrow \tau$, entonces el analizador ejecuta la reducción oportuna donde el nuevo estado en cabeza de la pila se obtiene mediante la función $\text{GOTO}(s_{m-r}, a_i) = s$ donde r es precisamente la longitud del consecuente τ. O sea, el analizador extrae primero $2 \cdot r$ símbolos de la pila (r

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

estados y los r símbolos de la gramática que cada uno tiene por debajo), exponiendo el estado s_{m-r} en la cima. Luego se introduce el no terminal A (antecedente de la regla aplicada), produciendo:

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_{m-r}\ s_{m-r}\ A \quad , \quad a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

A continuación se consulta la entrada (s_{m-r}, a_i) de la tabla GOTO: GOTO(s_{m-r}, A) = s_{m-r+1}, produciéndose finalmente:

$$(s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_{m-r}\ s_{m-r}\ A\ s_{m-r+1} \quad , \quad a_i\ a_{i+1}\ \ldots\ a_n\ \$)$$

donde s_{m-r+1} es el nuevo estado en la cima de la pila y no se ha producido variación en la subcadena de entrada que aún queda por analizar.

1.3.- Si ACCION(s_m, a_i) = ACEPTAR, entonces se ha llegado a la finalización en el proceso de reconocimiento y el análisis termina aceptando la cadena de entrada.

1.4-. Si ACCION(s_m, a_i) = RECHAZAR, entonces el analizador LR ha descubierto un error sintáctico y se debería proceder en consecuencia activando las rutinas de recuperación de errores. Como se ha comentado, una de las ventajas de este tipo de análisis es que cuando se detecta un error, el *token* erróneo suele estar al final de α o al principio de β, lo que permite depurar con cierta facilidad las cadenas de entrada (programas).

La aplicación de uno de estos cuatro pasos se aplica de manera reiterada hasta aceptar o rechazar la cadena (en caso de rechazo es posible continuar el reconocimiento si se aplica alguna técnica de recuperación de errores sintácticos). El punto de partida es la configuración:

$$(s_0 \quad , \quad a_1\ a_2\ \ldots\ a_n\ \$)$$

donde s_0 es el estado inicial del autómata.

Vamos a ilustrar con un ejemplo la aplicación de este método. Para ello partiremos de la gramática del cuadro 3.4. Dicha gramática tiene asociadas dos tablas una de ACCION y otra de GOTO. Por regla general, y por motivos de espacio, ambas tablas suelen fusionarse en una sola. Una tabla tal se divide en dos partes: unas columnas comunes a ACCION y GOTO, y otras columnas sólo de GOTO. Las casillas de las columnas que son sólo de GOTO contienen números de estados a los que se transita; si están vacías quiere decir que hay un error. Las columnas comunes a ambas tablas pueden contener:

① E	→	E + T
②	|	T
③ T	→	T * F
④	|	F
⑤ F	→	(E)
⑥	|	id

Cuadro 3.4 Gramática no ambigua que reconoce expresiones aritméticas en las que sólo intervienen identificadores

- D-i: significa "desplazar y pasar al estado i" (ACCION y GOTO todo en uno,

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

para ahorrar espacio).

- R j: significa "reducir por la regla de producción número j". En este caso debe aplicarse a continuación la tabla GOTO.
- Aceptar: la gramática acepta la cadena de terminales y finaliza el proceso de análisis.
- En blanco: rechaza la cadena y finaliza el proceso (no haremos recuperación de errores por ahora).

La tabla asociada a la gramática del cuadro 3.4 siguiendo los criterios anteriormente expuestos es:

	tabla ACCIÓN-GOTO						tabla GOTO		
ESTADO	id	+	*	(	)	$	E	T	F
0	D-5			D-4			1	2	3
1		D-6				Aceptar			
2		R2	D-7		R2	R2			
3		R4	R4		R4	R4			
4	D-5			D-4			8	2	3
5		R6	R6		R6	R6			
6	D-5			D-4				9	3
7	D-5			D-4					10
8		D-6			D-11				
9		R1	D-7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Supongamos ahora que se desea reconocer o rechazar la secuencia "id*(id+id)", y que el estado s_0 es, en nuestro caso, el 0. Así, la siguiente tabla muestra un ciclo del algoritmo por cada fila.

Pila	β	ACCION-GOTO
0	id * (id + id)$	D-5
0 id-5	* (id + id) $	R6-3
0 F-3	* (id + id) $	R4-2
0 T-2	* (id + id) $	D-7
0 T-2 *-7	(id + id) $	D-4
0 T-2 *-7 (-4	id + id) $	D-5
0 T-2 *-7 (-4 id-5	+ id) $	R6-3
0 T-2 *-7 (-4 F-3	+ id) $	R4-2
0 T-2 *-7 (-4 T-2	+ id) $	R2-8
0 T-2 *-7 (-4 E-8	+ id) $	D-6
0 T-2 *-7 (-4 E-8 +-6	id) $	D-5
0 T-2 *-7 (-4 E-8 +-6 id-5	) $	R6-3
0 T-2 *-7 (-4 E-8 +-6 F-3	) $	R4-9
0 T-2 *-7 (-4 E-8 +-6 T-9	) $	R1-8

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

0 T-2 *-7 (-4 E-8	) $	D-11
0 T-2 *-7 (-4 E-8) 11	$	R5-10
0 T-2 *-7 F-10	$	R3-2
0 T-2	$	R2-1
0 E-1	$	Aceptar

Nótese que con este método, la pila hace las funciones de α. La diferencia estriba en la existencia de estados intercalados: hay un estado inicial en la base de la pila, y cada símbolo de α tiene asociado un estado. Nótese, además, que cuando se reduce por una regla, el consecuente de la misma coincide con el extremo derecho de α.

3.5.6.1 Consideraciones sobre el análisis LR(1)

Uno de los primeros pasos que se deben dar durante la construcción de un traductor consiste en la adecuada selección de la gramática que reconozca el lenguaje. Y no es un paso trivial ya que, aunque por regla general existe una multitud de gramáticas equivalentes, cuando se trabaja con aplicaciones prácticas las diferencias de comportamientos entre gramáticas equivalentes adquiere especial relevancia.

En este apartado se dan un par de nociones sobre cómo elegir correctamente las gramáticas.

3.5.6.1.1 Recursión a derecha o a izquierda

Uno de los planteamientos más elementales consiste en decidir si utilizar reglas recursivas a izquierda o recursivas a derecha. Por ejemplo, para reconocer una secuencia de identificadores separados por comas como:

id, id, id

es posible optar por dos gramáticas diferentes:

a) ① lista → ID
② | lista ‘,’ ID

b) ① lista → ID
② | ID ‘,’ lista

Pues bien, a la hora de realizar el reconocimiento de una sentencia podemos observar que se obtienen árboles bien distintos, como en la figura 3.26 al reconocer “id, id, id”. A su vez, si aplicamos un reconocimiento LR, la secuencia de pasos aplicada difiere radicalmente: en el caso b) se desplazan secuencialmente, uno a uno todos los *tokens* de la cadena y, por último, se hacen varias reducciones seguidas, según la longitud de la cadena; en el caso a) se desplazan varios *tokens* (la coma y un ID) y se reduce por ②, y así sucesivamente se van alternando los desplazamientos y las reducciones.

De esta manera, con recursión a la derecha, la pila α va aumentando de tamaño hasta que comienzan las reducciones. Por tanto, ante listas de identificadores lo suficientemente grandes podría producirse un desbordamiento. Lo más conveniente, por tanto, es utilizar gramáticas recursivas a la izquierda para que el tamaño de la pila

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

se mantenga estable. En aquellos casos en que la recursión a la derecha sea absolutamente necesaria (algunos de estos casos se nos plantearán en capítulos posteriores) debe evitarse el suministro de cadenas excesivamente largas y que puedan producir desbordamientos.

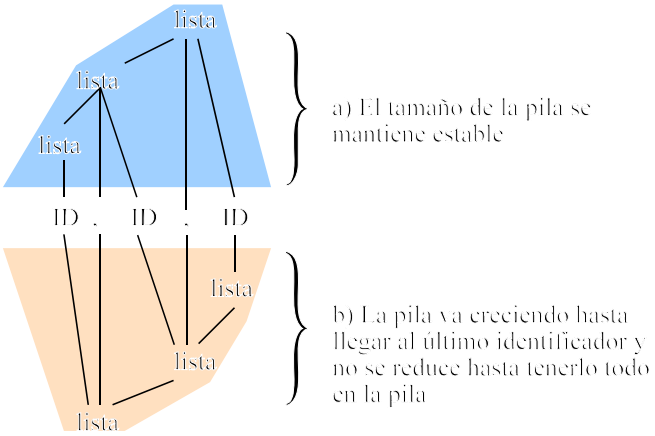


Figure 26 Reconocimiento de una misma sentencia por gramáticas equivalentes: a) recursiva a la izquierda, y b) recursiva a la derecha

3.5.6.1.2 Conflictos

Un conflicto se produce cuando el analizador no es capaz de generar la tabla de chequeo de sintaxis para una gramática. Básicamente pueden darse dos tipos de conflictos:

- Desplazar/Reducir (*Shift/Reduce*). Suelen deberse a gramáticas ambiguas.
- Reducir/Reducir (*Reduce/Reduce*). Debidos a que la gramática no admite un análisis LR(k).

El conflicto Desplazar/Reducir aparece cuando en la tabla de acciones aparece en la misma casilla tanto una R de reducir como una D de desplazar, el conflicto es que el programa no sabe si reducir o desplazar. Por ejemplo, una gramática tan sencilla

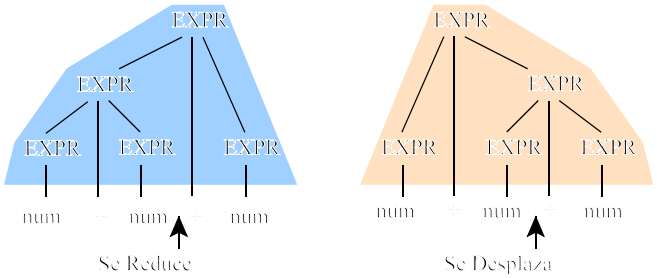


Figure 27 Ejemplo de ambigüedad

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

como:

EXPR → EXPR '+' EXPR
| num

resulta obviamente ambigua (es recursiva por la derecha y por la izquierda simultáneamente), luego ante una entrada tan sencilla como "num + num + num" pueden producirse dos árboles sintácticos diferentes (ver figura 3.27).

Por otro lado, el conflicto Reducir/Reducir aparece cuando el analizador puede aplicar dos o más reglas ante determinadas situaciones. Esta situación suele aparecer cuando la gramática no es LR(1), esto es, no es suficiente con un *token* de pre-búsqueda para tomar la decisión de qué acción realizar. Por ejemplo, supongamos la gramática:

① S → aaBdd
② | aCd
③ B → a
④ C → aa

que únicamente reconoce las dos secuencia de *tokens* "aaadd" y "aaad". Pues bien, una gramática tan sencilla no es LR(1) (ver figura 3.28). Supongamos que en esta figura se desea reconocer la secuencia "aaad"; en tal caso lo correcto sería reducir por ④. Sin embargo todo depende de si detrás del *token* de pre-búsqueda (una "d"), viene otra "d" o lleva EOF. Un analizador LR(2) sería capaz de tomar la decisión correcta, pero no uno LR(1).

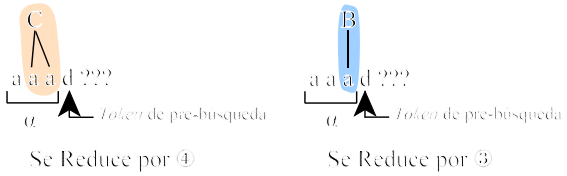


Figure 28 Ejemplo de conflicto Reduce/Reduce. Las interrogaciones indican que el analizador desconoce lo que viene a continuación

Los conflictos Reduce/Reduce se pueden eliminar en la mayoría de los casos. Esto se consigue retrasando la decisión de reducción al último momento. P.ej., la gramática:

S → aaadd
| aaad

es equivalente a la anterior, pero sí es LR(1), ya que sólo se decide reducir ante el EOF del final, en cuyo momento ya se estará en el estado correcto en función de los *tokens* que se han leído al completo.

3.5.6.2 Conclusiones sobre el análisis LR(1)

En el epígrafe 3.5.6 se estudió cómo funcionan los analizadores LR mediante

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

la utilización de sus correspondientes tablas de análisis. En el ejemplo final puede observarse como el estado que siempre se encuentra en cabeza de la pila contiene en todo momento la información necesaria para la reducción, si esto procede. Además, hemos dejado al margen intencionadamente la estructura del programa de proceso puesto que se trata esencialmente de un autómata finito con pila.

En general puede afirmarse que, dada la estructura de los analizadores LR, con la sola inspección de k símbolos de la cadena de entrada a la derecha del último consumido puede decidirse con toda exactitud cual es el movimiento a realizar (reducción, desplazamiento, etc). Es por este motivo por lo que suele denominarse a este tipo de gramáticas como LR(k). Como ya se comentó, en la práctica casi todos los lenguajes de programación pueden ser analizados mediante gramáticas LR(1).

Las tablas LR(1) ideadas por Knuth en 1965 son demasiado grandes para las gramáticas de los lenguajes de programación tradicionales. En 1969 De Remer y Korenjack descubrieron formas de compactar estas tablas, haciendo práctico y manejable este tipo de analizadores. El algoritmo que se aplica sobre dichas tablas es el mismo ya visto. También hemos visto que hay tres tipos principales de gramáticas LR(k) que varían en función de su potencia estando incluidas unas en otras de la siguiente forma:

$$SLR(k) \subset LALR(k) \subset LR(k)$$

pero no sus lenguajes respectivos que coinciden en sus conjuntos, esto es, si hay una gramática LR(1) que reconoce el lenguaje L, entonces también hay una gramática SLR(1) que reconoce a L.

Los metacompiladores Yacc y Cup utilizan el análisis LALR(1). El autómata debe de mantener información de su configuración (pila α), y para mantener información sobre β se comunica con Lex, quien se encarga de la metacompilación a nivel léxico.

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike

Open Course Ware
Gálvez Rojas, S.
(2011). Traductores, Compiladores e Intérpretes
OCW- Universidad de Málaga http://ocw.uma.es
Bajo licencia Creative Commons Attribution-Non-Comercial-ShareAlike
BY NC SA