INGENIERÍA EN SISTEMAS

Creamos, Transformamos y Simplificamos

Árboles Sintácticos

Ing. Alex Moncada

UNAH

UNIVERSIDAD NACIONAL
AUTÓNOMA DE HONDURAS

LU
CEM
ASPI
CIO

Derivaciones

La construcción de un árbol de análisis sintáctico puede hacerse precisa si tomamos una vista derivacional, en la cual las producciones se tratan como reglas de rescritura.

Como veremos, el análisis sintáctico ascendente se relaciona con una clase de derivaciones conocidas como derivaciones de “más a la derecha”, en donde el no terminal por la derecha se rescribe en cada paso.

Por ejemplo, considere la siguiente gramática, con un solo no terminal E, la cual agrega una producción E → − E a la gramática (4.3):

$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id} \qquad (4.7)$$

La producción E → − E significa que si E denota una expresión, entonces − E debe también denotar una expresión. La sustitución de una sola E por − E se describirá escribiendo lo siguiente: $E \Rightarrow -E$ lo cual se lee como “E deriva a − E ”.

Derivaciones

La producción E → (E) puede aplicarse para sustituir cualquier instancia de E en cualquier cadena de símbolos gramaticales por (E); por ejemplo, E ∗ E ⇒ (E) ∗ E o E ∗ E ⇒ E ∗ (E).

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$$

Para una definición general de la derivación, considere un no terminal A en la mitad de una secuencia de símbolos gramaticales, como en αAβ, en donde α y β son cadenas arbitrarias de símbolos gramaticales.

Suponga que A → γ es una producción. Entonces, escribimos αAβ ⇒ αγβ.

El símbolo ⇒ significa, “se deriva en un paso”.

Cuando una secuencia de pasos de derivación α1 ⇒ α2 ⇒ … ⇒ αn se rescribe como α1 a αn, decimos que α1 deriva a αn.

Con frecuencia es conveniente poder decir, “deriva en cero o más pasos”.

Para este fin, podemos usar el símbolo ∗⇒ . Así,

Derivaciones

1. $\alpha \overset{*}{\Rightarrow} \alpha$, para cualquier cadena α.

2. Si $\alpha \overset{*}{\Rightarrow} \beta$ y $\beta \Rightarrow \gamma$, entonces $\alpha \overset{*}{\Rightarrow} \gamma$.

De igual forma, ⇒ significa “deriva en uno o más pasos”. Si S ∗⇒ α, en donde S es el símbolo inicial de una gramática G, decimos que α es una forma de frase de G. Observe que una forma de frase puede contener tanto terminales como no terminales, y puede estar vacía. Un enunciado de G es una forma de frase sin símbolos no terminales.

El lenguaje generado por una gramática es su conjunto de oraciones. Por ende, una cadena de terminales *w* está en L(G), el lenguaje generado por G, si y sólo si *w* es un enunciado de G (o S ∗⇒ *w*).

Un lenguaje que puede generarse mediante una gramática se considera un lenguaje libre de contexto. Si dos gramáticas generan el mismo lenguaje, se consideran como equivalentes.

Derivaciones

La cadena −(id + id) es un enunciado de la gramática (4.7), ya que hay una derivación

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id}) \qquad (4.8)$$

Para comprender la forma en que trabajan los analizadores sintácticos, debemos considerar las derivaciones en las que el no terminal que se va a sustituir en cada paso se elige de la siguiente manera:

1. En las derivaciones por la izquierda, siempre se elige el no terminal por la izquierda en cada de frase. Si α ⇒ β es un paso en el que se sustituye el no terminal por la izquierda en α, escribimos α ⇒ lm β.

2. En las derivaciones por la derecha, siempre se elige el no terminal por la derecha; en este caso escribimos α ⇒ rm β.

La derivación (4.8) es por la izquierda, por lo que puede rescribirse de la siguiente manera:

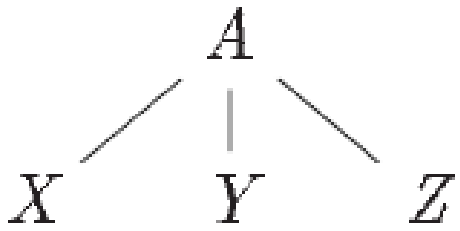
$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + E) \underset{lm}{\Rightarrow} -(\mathbf{id} + \mathbf{id})$$

Derivaciones

Si utilizamos nuestras convenciones de notación, cada paso por la izquierda puede escribirse como wAγ ⇒lmwδγ, en donde w consiste sólo de terminales, A → δ es la producción que se aplica, y **γ** es una cadena de símbolos gramaticales. Para enfatizar que α deriva a β mediante una derivación por la izquierda, escribimos α∗⇒lmβ. Si S∗⇒lmα, decimos que α es una forma de frase izquierda de la gramática en cuestión.

Árboles de análisis sintáctico

Un árbol de análisis sintáctico muestra, en forma gráfica, la manera en que el símbolo inicial de una gramática deriva a una cadena en el lenguaje. Si el *no terminal* **A** tiene una producción A → XYZ, entonces un árbol de análisis sintáctico podría tener un nodo interior etiquetado como A, con tres hijos llamados X, Y y Z, de izquierda a derecha:



Árboles de análisis sintáctico

De manera formal, dada una gramática libre de contexto, un árbol de análisis sintáctico de acuerdo con la gramática es un árbol con las siguientes propiedades:

1. La raíz se etiqueta con el símbolo inicial.
2. Cada hoja se etiqueta con *un terminal*, o con ***e***
3. Cada nodo interior se etiqueta con un *no terminal.*
4. Si A es el no terminal que etiqueta a cierto nodo interior, y X1, X2, . . ., Xn son las etiquetas de los hijos de ese nodo de izquierda a derecha, entonces debe haber una producción A→X1X2 · · · Xn. Aquí, cada una de las etiquetas X1, X2, . . ., Xn representa a un símbolo que puede ser o no un terminal. Como un caso especial, si A → ***e*** es una producción, entonces un nodo etiquetado como A puede tener un solo hijo, etiquetado como **e**

Terminología de árboles

Las estructuras de datos tipo árbol figuran de manera prominente en la compilación.

- Un árbol consiste en uno o más nodos. Los nodos pueden tener etiquetas, serán símbolos de la gramática. Al dibujar un árbol, con frecuencia representamos los nodos mediante estas etiquetas solamente.

- Sólo uno de los nodos es la raíz. Todos los nodos, excepto la raíz, tienen un padre único; la raíz no tiene padre. Al dibujar árboles, colocamos el padre de un nodo encima de ese nodo y dibujamos una línea entre ellos. Entonces, la raíz es el nodo más alto (superior).

Terminología de árboles

- Si el nodo N es el padre del nodo M, entonces M es hijo de N. Los hijos de nuestro nodo se llaman hermanos. Tienen un orden, partiendo desde la izquierda, por lo que al dibujar árboles, ordenamos los hijos de un nodo dado en esta forma.

- Un nodo sin hijos se llama hoja. Los otros nodos (los que tienen uno o más hijos) son nodos interiores.

- Un descendiente de un nodo N es ya sea el mismo N, un hijo de N, un hijo de un hijo de N, y así en lo sucesivo, para cualquier número de niveles.

Ejemplo de árboles

Ejemplo 2.4: La derivación de 9−5+2 en el ejemplo 2.2 se ilustra mediante el árbol en la figura 2.5. Cada nodo en el árbol se etiqueta mediante un símbolo de la gramática. Un nodo interior y su hijo corresponden a una producción; el nodo interior corresponde al encabezado de la producción, el hijo corresponde al cuerpo.

En la figura 2.5, la raíz se etiqueta como lista, el símbolo inicial de la gramática en el ejemplo 2.1. Los hijos de la raíz se etiquetan, de izquierda a derecha, como lista, + y dígito. Observe que es una producción en la gramática del ejemplo 2.1. El hijo izquierdo de la raíz es similar a la raíz, con un hijo etiquetado como − en vez de +. Los tres nodos etiquetados como dígito tienen cada uno un hijo que se etiqueta mediante un dígito.

$$lista \rightarrow lista + dígito$$

Ejemplo de árboles

De izquierda a derecha, las hojas de un árbol de análisis sintáctico forman la derivación del árbol: la cadena que se genera o deriva del no terminal en la raíz del árbol de análisis sintáctico.

En la figura 2.5, la derivación es 9−5+2; por conveniencia, todas las hojas se muestran en el nivel inferior. Por lo tanto, no es necesario alinear las hojas de esta forma. Cualquier árbol imparte un orden natural de izquierda a derecha a sus hojas, con base en la idea de que si X y Y son dos hijos con el mismo padre, y X está a la izquierda de Y, entonces todos los descendientes de X están a la izquierda de los descendientes de Y. Otra definición del lenguaje generado por una gramática es como el conjunto de cadenas que puede generar cierto árbol de análisis sintáctico. Al proceso de encontrar un árbol de análisis sintáctico para una cadena dada de terminales se le llama analizar sintácticamente esa cadena.

Ejemplo de árboles

$$
\begin{aligned}
\textit{lista} &\rightarrow \textit{lista} + \textit{dígito} \\
\textit{lista} &\rightarrow \textit{lista} - \textit{dígito} \\
\textit{lista} &\rightarrow \textit{dígito} \\
\textit{dígito} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

9 - 5 + 2

Figura 2.5: Árbol de análisis sintáctico para 9−5+2, de acuerdo con la gramática en el ejemplo 2.1

Ambigüedad

- Tenemos que ser cuidadosos al hablar sobre la estructura de una cadena, de acuerdo a una gramática.
- Una gramática puede tener más de un árbol de análisis sintáctico que genere una cadena dada de terminales. Se dice que dicha gramática es ambigua.
- Para mostrar que una gramática es ambigua, todo lo que debemos hacer es buscar una cadena de terminales que sea la derivación de más de un árbol de análisis sintáctico.
- Como una cadena con más de un árbol de análisis sintáctico tiene, por lo general, más de un significado, debemos diseñar gramáticas no ambiguas para las aplicaciones de compilación, o utilizar gramáticas ambiguas con reglas adicionales para resolver las ambigüedades.

Ambigüedad

Ejemplo 2.5: Suponga que utilizamos una sola cadena no terminal y que no diferenciamos entre los dígitos y las listas, como en el ejemplo 2.1. Podríamos haber escrito la siguiente gramática:

$$cadena \rightarrow cadena + cadena \mid cadena - cadena \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Mezclar la noción de dígito y lista en la cadena no terminal tiene sentido superficial, ya que un dígito individual es un caso especial de una lista. No obstante, la figura 2.6 muestra que una expresión como 9−5+2 tiene más de un árbol de análisis sintáctico con esta gramática. Los dos árboles para 9−5+2 corresponden a las dos formas de aplicar paréntesis a la expresión: (9−5)+2 y 9−(5+2). Este segundo uso de los paréntesis proporciona a la expresión el valor inesperado de 2, en vez del valor ordinario de 6. La gramática del ejemplo 2.1 no permite esta interpretación.

Ambigüedad

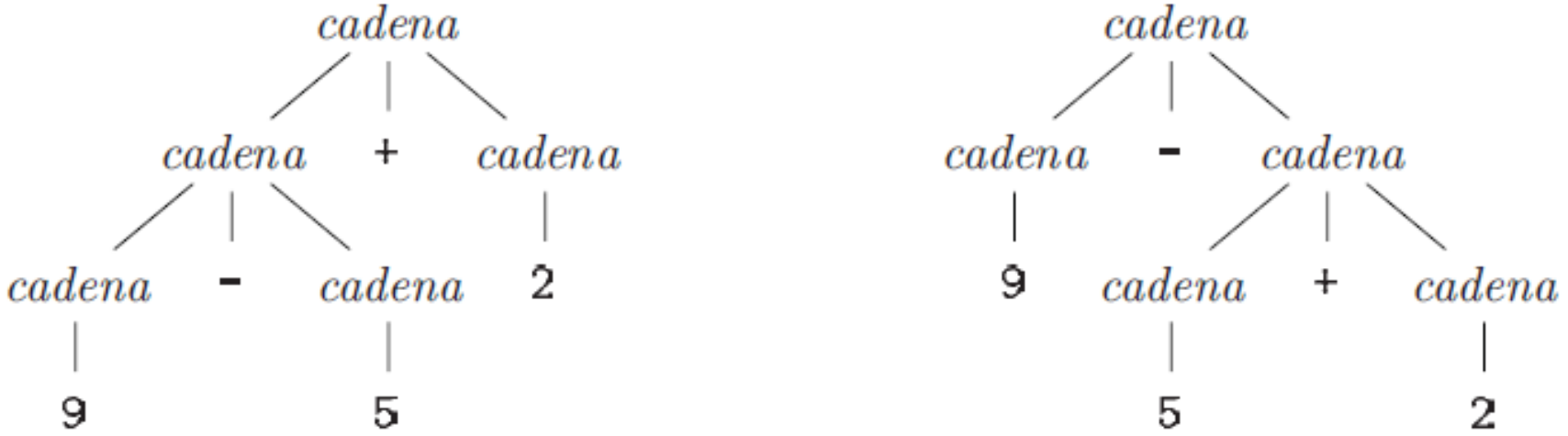


Figura 2.6: Dos árboles de análisis sintáctico para 9−5+2

Asociatividad de los operadores

- Por convención, 9+5+2 es equivalente a (9+5)+2 y 9−5−2 es equivalente a (9−5)−2. Cuando un operando como 5 tiene operadores a su izquierda y a su derecha, se requieren convenciones para decidir qué operador se aplica a ese operando.

- Decimos que el operador + se asocia por la izquierda, porque un operando con signos positivos en ambos lados de él pertenece al operador que está a su izquierda.

- En la mayoría de los lenguajes de programación, los cuatro operadores aritméticos (suma, resta, multiplicación y división) son asociativos por la izquierda.

Asociatividad de los operadores

Algunos operadores comunes, como la exponenciación, son asociativos por la derecha. Como otro ejemplo, el operador de asignación = en C y sus descendientes es asociativo por la derecha; es decir, la expresión a=b=c se trata de la misma forma que la expresión a=(b=c).

Las cadenas como a=b=c con un operador asociativo por la derecha se generan mediante la siguiente gramática:

$$\begin{aligned} \textit{derecha} &\rightarrow \textit{letra} = \textit{derecha} \mid \textit{letra} \\ \textit{letra} &\rightarrow \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \end{aligned}$$

El contraste entre un árbol de análisis sintáctico para un operador asociativo por la izquierda como −, y un árbol de análisis sintáctico para un operador asociativo por la derecha como =, se muestra en la figura 2.7. Observe que el árbol de análisis sintáctico para 9−5−2 crece hacia abajo y a la izquierda, mientras que el árbol de análisis sintáctico para a=b=c crece hacia abajo y a la derecha.

Asociatividad de los operadores

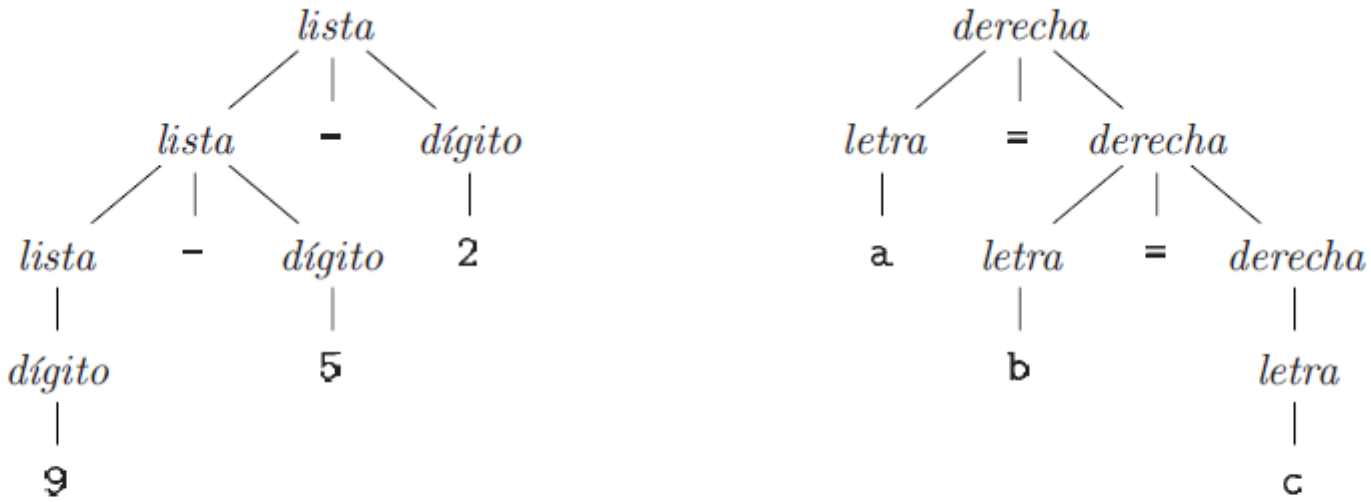


Figura 2.7: Árboles de análisis sintáctico para las gramáticas asociativas por la izquierda y por la derecha

Precedencia de operadores

Considere la expresión 9+5*2. Hay dos posibles interpretaciones de esta expresión: (9+5)*2 o 9+(5*2). Las reglas de asociatividad para + y * se aplican a las ocurrencias del mismo operador, por lo que no resuelven esta ambigüedad. Las reglas que definen la precedencia relativa de los operadores son necesarias cuando hay más de un tipo de operador presente.

Decimos que * tiene mayor precedencia que +, si * recibe sus operandos antes que +. En la aritmética ordinaria, la multiplicación y la división tienen mayor precedencia que la suma y la resta. Por lo tanto, * recibe el 5 tanto en 9+5*2 como en 9*5+2; es decir, las expresiones son equivalentes a 9+(5*2) y (9*5)+2, respectivamente.

Precedencia de operadores

Ejemplo 2.6: Podemos construir una gramática para expresiones aritméticas a partir de una tabla que muestre la asociatividad y la precedencia de los operadores. Empezamos con los cuatro operadores aritméticos comunes y una tabla de precedencia, mostrando los operadores en orden de menor a mayor precedencia. Los operadores en la misma línea tienen la misma asociatividad y precedencia:

asociativo por la izquierda:	+ -
asociativo por la derecha:	* /

Creamos dos no terminales llamadas expr y term para los dos niveles de precedencia, y un no terminal adicional llamado factor para generar unidades básicas en las expresiones. Las unidades básicas en las expresiones son dígitos y expresiones entre paréntesis.

$$\textit{factor} \rightarrow \textbf{dígito} \mid (\ \textit{expr}\ )$$

Precedencia de operadores

Ahora consideremos los operadores binarios, * y /, que tienen la mayor precedencia. Como estos operadores asocian por la izquierda, las producciones son similares a las de las listas que asocian por la izquierda.

$$\begin{array}{rcl} term & \rightarrow & term * factor \\ & | & term \,/\, factor \\ & | & factor \end{array}$$

De manera similar, expr genera listas de términos separados por los operadores aditivos:

$$\begin{array}{rcl} expr & \rightarrow & expr + term \\ & | & expr - term \\ & | & term \end{array}$$

Por lo tanto, la gramática resultante es:

$$\begin{array}{rcl} expr & \rightarrow & expr + term \mid expr - term \mid term \\ term & \rightarrow & term * factor \mid term \,/\, factor \mid factor \\ factor & \rightarrow & \textbf{dígito} \mid (\ expr\ ) \end{array}$$

Con esta gramática, una expresión es una lista de términos separados por los signos + o −, y un término es una lista de factores separados por los signos * o /.

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:

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

Manejo de errores sintácticos

- 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.

Manejo de errores sintácticos

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.

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).

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

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

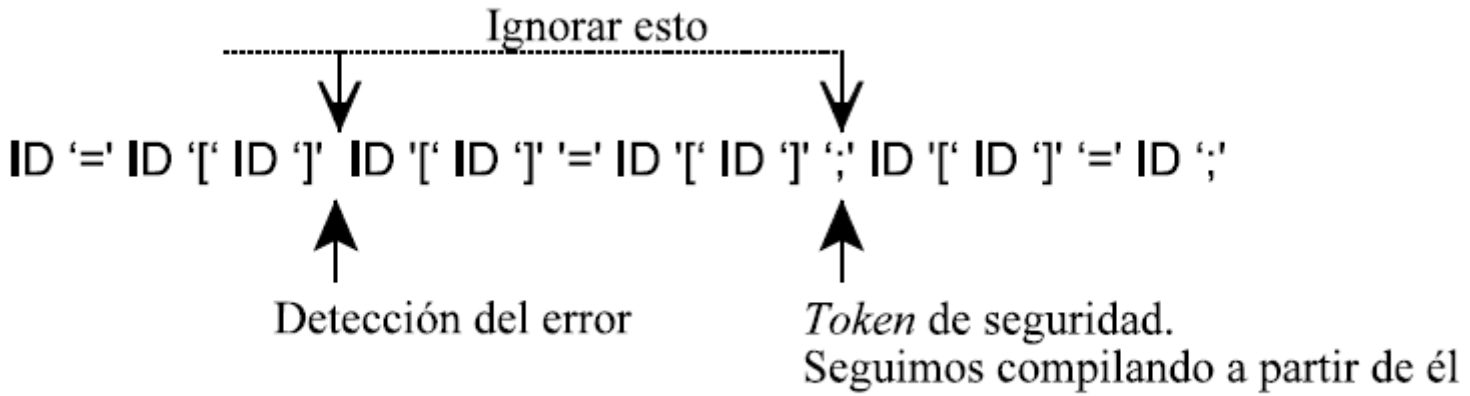


Figura 3.1. Recuperación de error sintáctico ignorando la secuencia errónea

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.

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 anterior, 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.

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.

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.
- 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.

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. 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.

Análisis descendente de gramáticas 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. 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×T**Ư**{$}].
- Como siempre, la cadena de entrada acabará en EOF que, a partir de ahora, denotaremos por el símbolo '$'.

Análisis descendente de gramáticas LL(1)

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 != $ entonces

- se quita X de la pila
- y se avanza el apuntador.

Análisis descendente de gramáticas LL(1)

3.- Si X e T y X .!= a entonces RECHAZAR.

4.- Si X e 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.

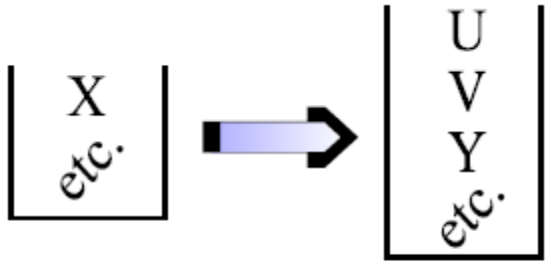


Figura 3.19. Aplicación de regla en el método LL(1)

Análisis descendente de gramáticas LL(1)

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’ | **e**

T -> F T’

T’ -> * F T’ | **e**

F -> (E) | id

Análisis descendente de gramáticas LL(1)

su tabla M:

N \ T	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)		

Análisis descendente de gramáticas LL(1)

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.

Análisis descendente de gramáticas LL(1)

E -> T E'

E' -> + T E' | **e**

T -> F T'

T' -> * F T' | **e**

F -> (E) | id

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)		

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

Generalidades del análisis ascendente

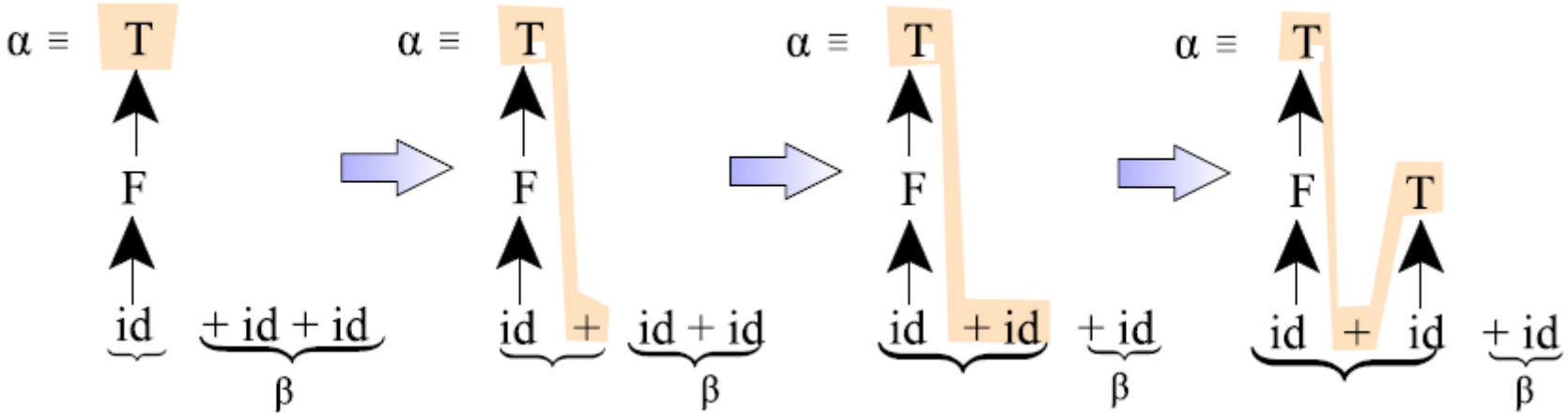
El objetivo de un análisis ascendente consiste en construir el árbol sintáctico desde abajo hacia arriba, esto es, desde los tokens hacia el axioma inicial, lo cual disminuye el número de reglas mal aplicadas con respecto al caso descendente (si hablamos del caso con retroceso) o amplia el número de 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: β **e** 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: α **e** (N u T)*.

Generalidades del análisis ascendente



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:

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:

Operaciones en un analizador ascendente

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

(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:

Operaciones en un analizador ascendente

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

un desplazamiento pasaría a:

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

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.