ANALISIS SINTACTICO

INTRODUCCION

Todo lenguaje de programación tiene reglas que prescriben la estructura sintáctica de programas bien formados. En Pascal, por ejemplo, un programa se compone de bloques, un bloque de proposiciones, una proposición de expresiones, una expresión de componentes léxicos, y así sucesivamente.

Se puede describir la sintaxis de las construcciones de los lenguajes de programación por medio de gramáticas independientes del contexto o notación BNF (Forma de Backus – Naur). Las gramáticas ofrecen ventajas significativas a los diseñadores de lenguajes y programadores de compiladores.

Una gramática da una especificación sintáctica precisa y fácil de entender de un lenguaje de programación.

A partir de algunas clases de gramáticas se puede construir automáticamente un analizador sintáctico eficiente que determine si un programa fuente está sintácticamente bien formado.

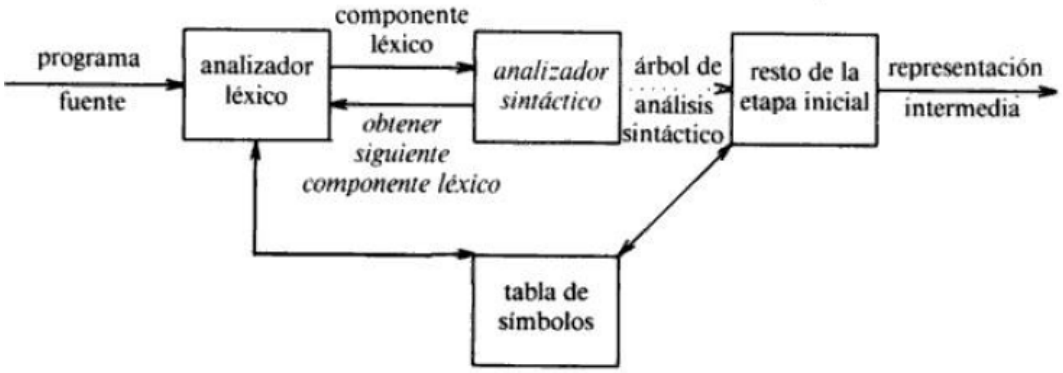
Una gramática diseñada adecuadamente imparte una estructura a un lenguaje de programación útil para la traducción de programas fuente a código objeto correcto y para la detección de errores.

Los lenguajes evolucionan con el tiempo, adquiriendo nuevas construcciones y realizando tareas adicionales. Estas nuevas construcciones se pueden añadir con más facilidad a un lenguaje cuando existe una aplicación basada en una descripción gramatical del lenguaje

EL PAPEL DEL ANALIZADOR SINTACTICO

En este modelo de compilador, el analizador sintáctico obtiene una cadena de componentes léxicos del analizador léxico, como se muestra en la figura y comprueba si la cadena pueda ser generada por la gramática del lenguaje fuente. Se supone que el analizador sintáctico informará de cualquier error de sintaxis de

manera inteligible. También debería recuperarse de los errores que ocurren frecuentemente para poder continuar procesando el resto de su entrada.



Los métodos empleados generalmente en los compiladores se clasifican como descendentes o ascendentes. Como sus nombres indican, los analizadores sintácticos descendentes construyen árboles de análisis sintáctico desde arriba (la raíz) hasta abajo (las hojas), mientras que los analizadores sintácticos ascendentes comienzan en las hojas y suben hacia la raíz. En ambos casos, se examina la entrada al analizador sintáctico de izquierda a derecha, un símbolo a la vez.

Los métodos descendentes y ascendentes más eficientes trabajan sólo con subclases de gramáticas, pero varias de estas subclases, como las gramáticas LL y LR, son lo suficientemente expresivas para describir la mayoría de las construcciones sintácticas de los lenguajes de programación. Los analizadores sintácticos implantados a mano a menudo trabajan con gramáticas LL1; por ejemplo, el método de la sección 2.4 construye analizadores sintácticos para gramáticas LL1. Los analizadores sintácticos para la clase más grande de gramáticas LR se construyen normalmente con herramientas automatizadas.

Manejo de Errores

Si un compilador tuviera que procesar solo programa correctos, su diseño e implantación se simplificarían mucho. Pero los programadores a menudo escriben programas incorrectos y un buen compilador debería ayudar al programador a identificar y localizar errores

Se sabe que los programas pueden contener errores de muy diverso tipo. Por ejemplo, los errores pueden ser:

- Léxicos, como escribir mal un identificador, palabra clave u operador.
- Sintácticos, como una expresión aritmética con paréntesis no equilibrados.
- Semánticos, como un operador aplicando a un operando incompatible.
- Lógicos, como una llamada infinitamente recursiva.

A menudo, gran parte de la detección y recuperación en un compilador se centra en la fase de análisis sintáctico. Una razón es que muchos errores son de naturaleza sintáctica o se manifiestan cuando la cadena de componentes léxicos que proviene de un analizador léxico desobedece las reglas gramaticales que definen al lenguaje de programación.

El manejador de errores en un analizador sintáctico tiene objetivos fáciles de establecer:

- Debe informar de la presencia de errores con claridad y exactitud.
- Se debe recuperar de cada error con la suficiente rapidez como para detectar errores posteriores.
- No debe retrasar de manera significativa el procesamiento de programas correctos.

Varios métodos de análisis sintáctico, como los métodos LL y LR, detectan un error lo antes posible. Es decir, tienen la propiedad del prefijo viable, lo cual quiere decir que detectan la presencia de un error nada más ver un prefijo de la entrada que no es prefijo de ninguna cadena del lenguaje.

Gramática Libre de Contexto

Muchas construcciones de los lenguajes de programación tienen una estructura inherentemente recursiva que se puede definir mediante gramáticas independientemente del contexto. Por ejemplo, se puede tener una proposición condicional definida por una regla como:

Si *S1* y *S2* son proposiciones y E es una expresión, entonces:

"if *E* then *S1* else *S2*" es una proposición.

No se puede especificar esta forma de proposición condicional usando la notación de las expresiones regulares .

Si utilizamos la variable sintáctica prop para denotar la clase de las proposiciones y expr para la clase de las expresiones, ya se puede expresar la proposición usando la producción gramatical.

prop ⟶ **if** *expr* **then** *prop* **else** *prop*

1. Los terminales son los símbolos básicos con que se forman las cadenas. "Componente Léxico" es un sinónimo de "terminal" cuando se trata de gramáticas para lenguajes de programación. (**if, then, else**).
2. Los no terminales son variables sintácticas que denotan conjuntos de cadenas. (*prop y expr* son no terminales). Los no terminales definen conjuntos de cadenas que ayudan a definir el lenguajes generado por la gramatica.
3. En una gramática, un no terminal es considerado como el símbolo inicial, y el conjunto de cadenas que representa es el lenguaje definido por la gramática.
4. Las producciones de una gramática especifican como se pueden combinar los terminales y los no terminales para formar cadenas. Cada producción consta de un no terminal, seguido por una flecha (a veces se usa el símbolo ::==, en lugar de la flecha), seguida por una cadena de no terminales y terminales.

Convenciones de Notación

Para evitar tener que establecer siempre que "estos son los terminales", "estos son los no terminales", etc., a partir de ahora se emplearán las siguientes convenciones de notaciones con respecto a las gramáticas:

1. Estos símbolos son terminales:
    a. Las primeras letras minúsculas del alfabeto, como a, b, c.
    b. Los símbolos de los operador, como +, -, etc.
    c. Los símbolos de puntuación, como paréntesis, coma, etc.
    d. Los dígitos 0, 1, … 9.
    e. Cadenas en negritas como **id** o **if.**
2. Estos símbolos son no terminales:
    a. Las primeras letras mayúsculas del alfabeto, como A, B, C.
    b. La letra *S*, que cuando aparece suele ser el símbolo inicial.
    c. Los nombres en cursivas minúsculas, como *expr* o *prop*.
3. Las últimas letras minúsculas del alfabeto, principalmente *u,v,…., z*. representan cadenas de terminales.
4. Las últimas letras mayúsculas del alfabeto como *X, Y, Z*, representan símbolos gramaticales, es decir, terminales o no terminales.
5. Las letras griegas minúsculas α, β, γ, Por ejemplo, representan símbolos gramaticales.

Ejemplo: La gramática con las siguientes producciones define expresiones aritméticas simples:

$$
\begin{aligned}
expr &\rightarrow expr\ op\ expr \\
expr &\rightarrow (\ expr\ ) \\
expr &\rightarrow -\ expr \\
expr &\rightarrow \mathbf{id} \\
op &\rightarrow + \\
op &\rightarrow - \\
op &\rightarrow * \\
op &\rightarrow / \\
op &\rightarrow \uparrow
\end{aligned}
$$

En esta gramática los símbolos terminales son: **id** + − * / ↑ ( )

Y los no terminales son: *expr y op* y *expr* es el símbolo inicial

Ejemplo: Usando estas abreviaturas se podría en forma concisa la gramática del ejemplo anterior.

$$
\begin{aligned}
E &\rightarrow E\ A\ E \mid (\ E\ ) \mid -\ E \mid \mathbf{id} \\
A &\rightarrow + \mid - \mid * \mid / \mid \uparrow
\end{aligned}
$$

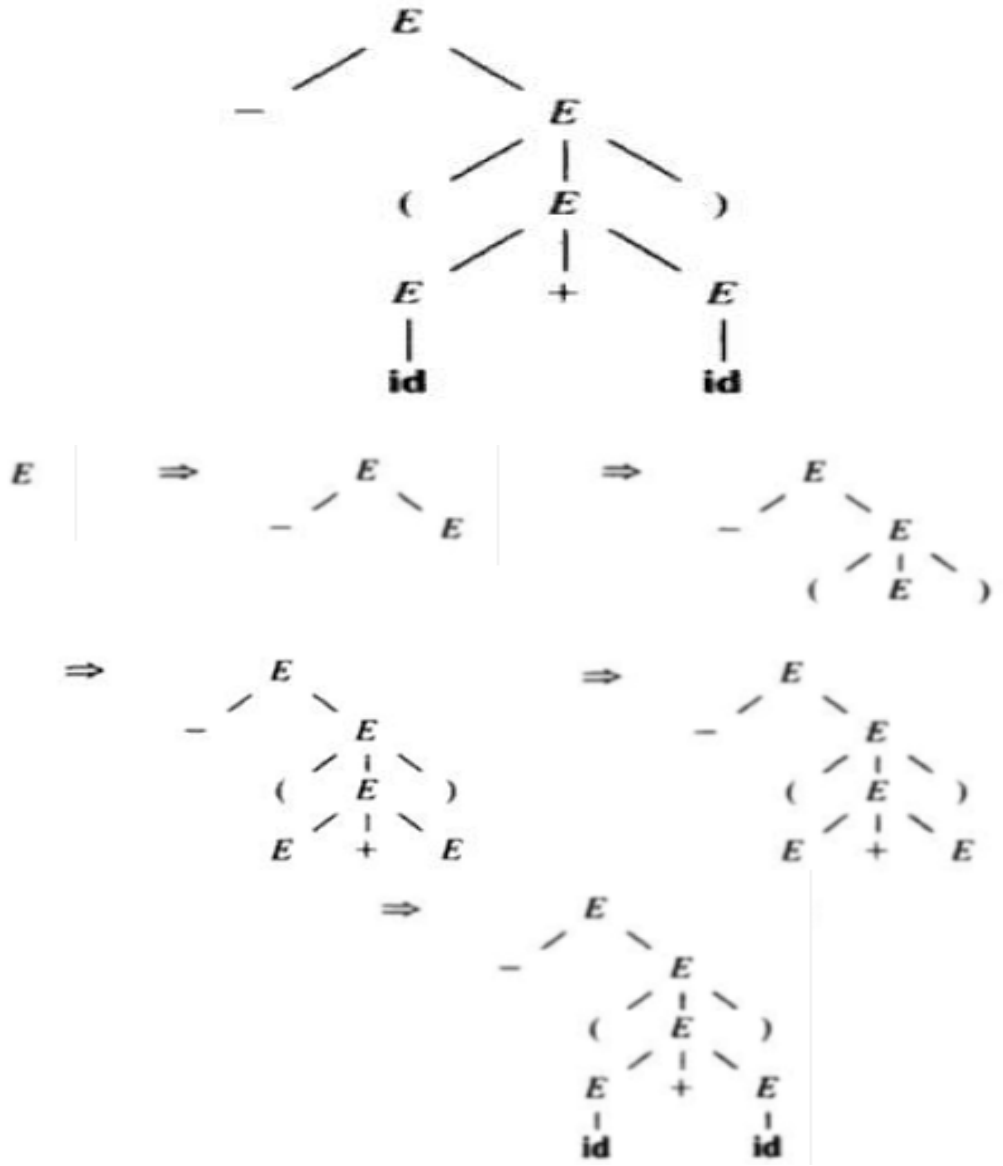
Las convenciones de notación indican que *E* y *A* son no terminales, con *E* como símbolo inicial. El resto de símbolos son terminales.

Ejemplo: La cadena – (id + id) es una frase de la gramática anterior, porque existe la derivación.

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

Árbol de Análisis Sintáctico y Derivaciones

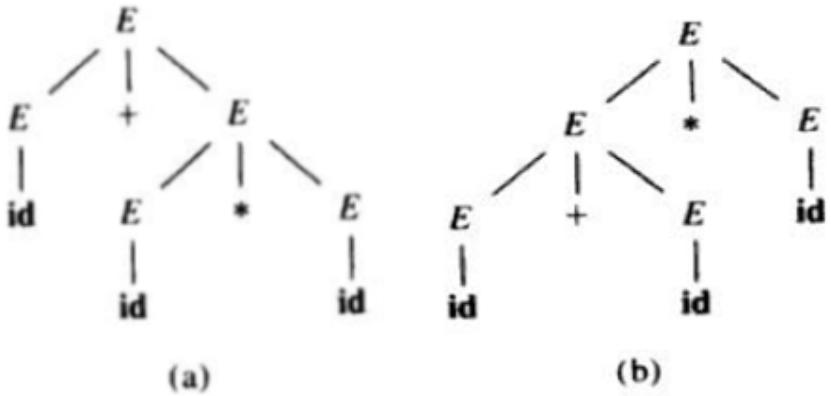
Un árbol de análisis sintáctico se puede considerar como una representación gráfica de una derivación que no muestra la elección relativa al orden de sustitución.



Ejemplo: Se considera de nuevo la gramática anterior de expresiones aritméticas. La frase id + id * id tiene las dos claras derivaciones por la izquierda.

$$E \Rightarrow E+E \Rightarrow \mathbf{id}+E \Rightarrow \mathbf{id}+E*E \Rightarrow \mathbf{id}+\mathbf{id}*E \Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}$$

$$E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow \mathbf{id}+E*E \Rightarrow \mathbf{id}+\mathbf{id}*E \Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}$$

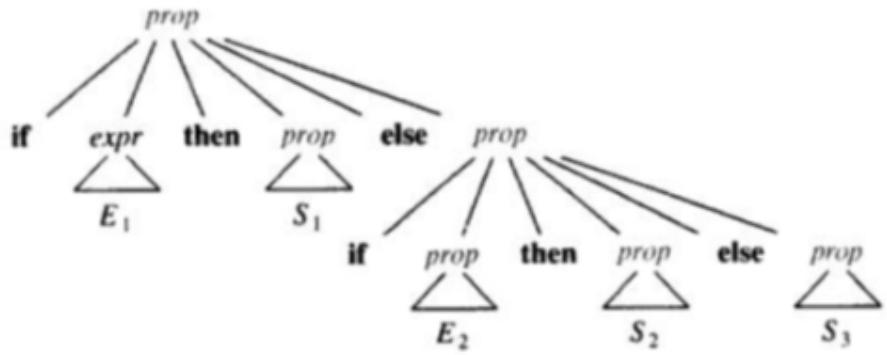


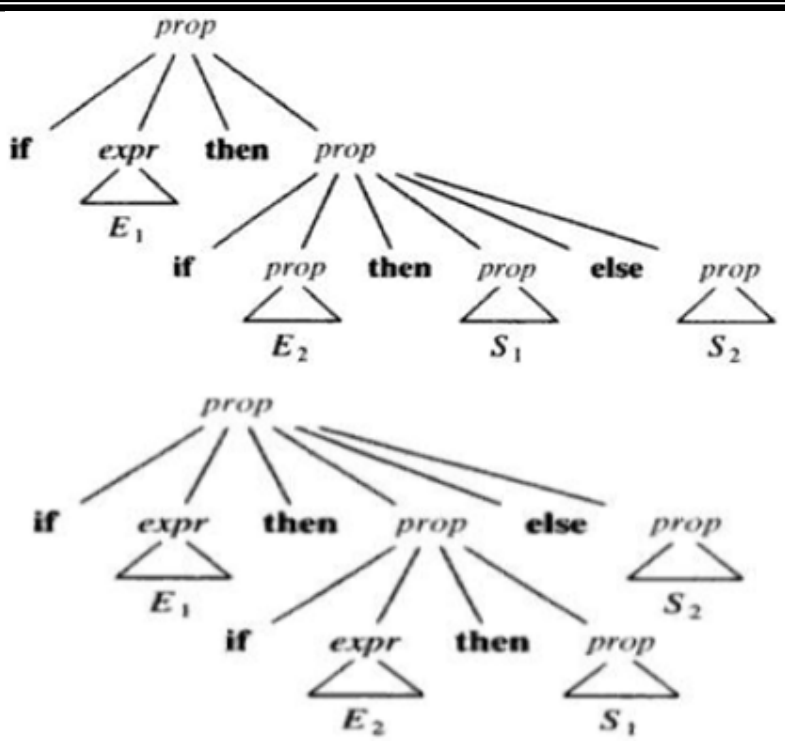
(a) (b)

Ambigüedad:

Se dice que una gramática que produce más de un árbol sintáctico para alguna frase es ambigua. O dicho de otro modo, una gramática ambigua es la que produce mas de una derivación por la izquierda o por la derecha de la misma frase.

$$prop \rightarrow \textbf{if } expr \textbf{ then } prop \mid \textbf{if } expr \textbf{ then } prop \textbf{ else } prop \mid \textbf{otra}$$

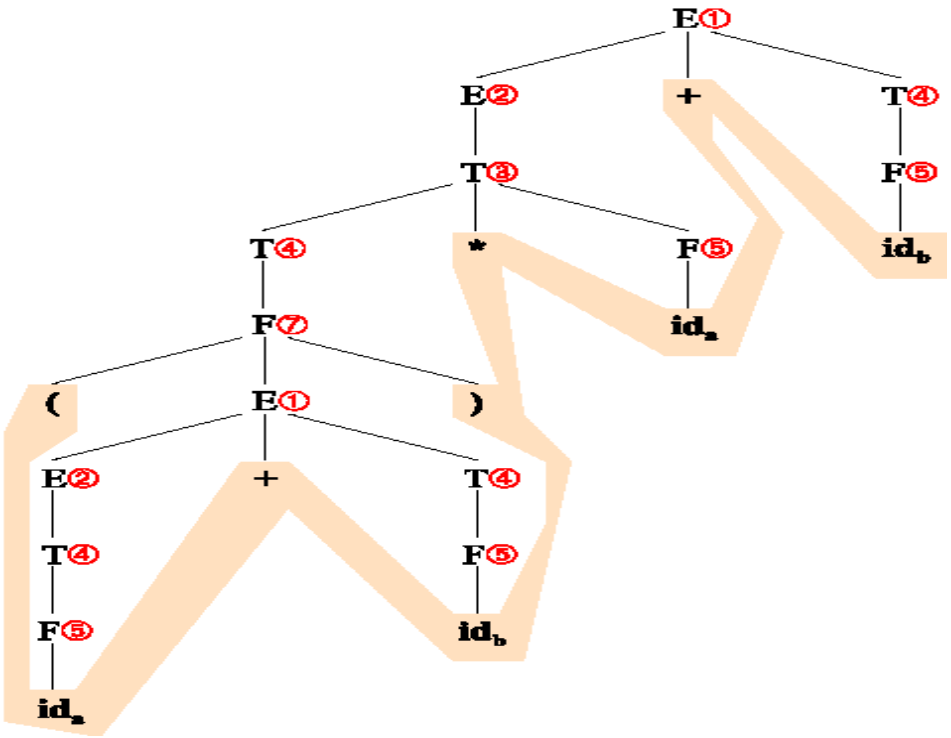




En todos los lenguajes de programación con proposiciones condicionales, se prefiere el primer árbol sintáctico. La regla general es “emparejar cada else con el then sin emparejar anterior más cercano”. Esta regla para eliminar ambigüedades se puede incorporar directamente a la gramática.

$$
\begin{aligned}
\textit{prop} &\rightarrow \textit{prop\_emparejada} \\
&\mid \textit{pro\_no\_emparejada} \\
\textit{prop\_emparejada} &\rightarrow \textbf{if } \textit{expr } \textbf{then } \textit{prop\_emparejada } \textbf{else } \textit{prop\_emparejada} \\
&\mid \textbf{otra} \\
\textit{prop\_no\_emparejada} &\rightarrow \textbf{if } \textit{expr } \textbf{then } \textit{prop} \\
&\mid \textbf{if } \textit{expr } \textbf{then } \textit{prop\_emparejada } \textbf{else } \textit{prop\_no\_emparejada}
\end{aligned}
$$

Análisis Sintáctico Descendente



Análisis Sintáctico Descendente con Retroceso

El análisis sintáctico descendente (ASD) intenta encontrar entre las producciones de la gramática la derivación por la izquierda del símbolo inicial para una cadena de entrada.

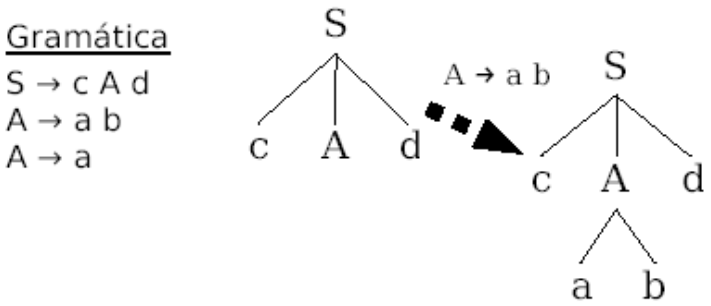
Ejemplo:

Analizar la cadena de entrada "cad" dada la gramática siguiente:

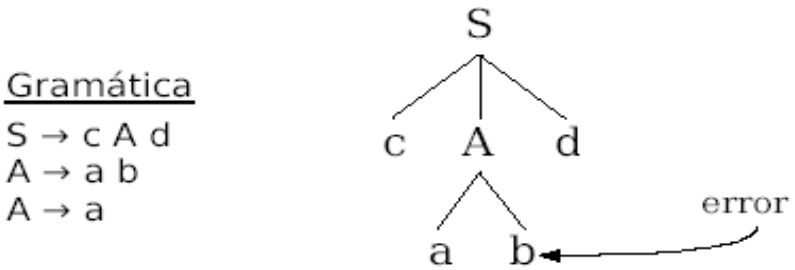
$$S \rightarrow c\ A\ d$$
$$A \rightarrow a\ b \mid a$$

"cad", se toma la primera producción

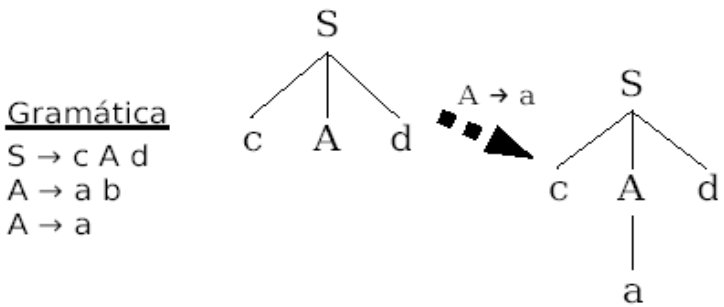
- "cad", se toma la segunda producción.
- siguiente hoja del árbol A → cabd



- "cad" se compara con la siguiente hoja del árbol etiquetada con "b". Como no concuerda, se indica el error y se vuelve a A para ver si hay otra alternativa no intentada.



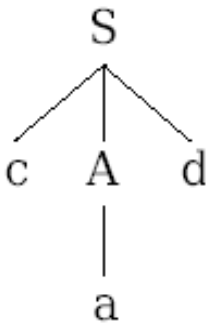
- "cad", se toma la siguiente alternativa que comienza por a.
- siguiente hoja del árbol A, → cad



"cad", coincide d con d → análisis exitoso

Gramática

S → c A d
A → a b
A → a



Análisis Sintáctico Descendente con Predictivo

El analizador debe realizar la previsión de la regla a aplicar sólo con ver el primer símbolo que produce para que el algoritmo tenga una complejidad lineal.

Ejemplo:

. Sent **if** Express then Sent

. Sent **while** Express do Sent

. Sent **begin** Sent end

Existe sólo una posibilidad de derivación, según que el primer símbolo que haya en la entrada sea un if, while o begin

Análisis Sintáctico Descendente con Predictivo

Las gramáticas que son susceptibles de ser analizadas sintácticamente de forma descendente mediante un análisis predictivo y consultando un únicamente un símbolo de entrada pertenecen al grupo LL(1).

A partir de gramáticas LL(1) se pueden construir analizadores sintácticos descendentes predictivos (ASDP), que son ASD sin retroceso.

Conjuntos de Predicción

- Son conjuntos de símbolos terminales
  - Ayudan a predecir qué regla se debe aplicar para el no Terminal que hay que derivar.
- Se construyen a partir de los símbolos de las partes derechas de las producciones de la gramática.
- El analizador consulta el siguiente símbolo en la entrada.
  - si pertenece al conjunto de predicción de una regla aplica esa regla, si no da error.

Ejemplos de Conjuntos de Predicción

Supóngase la entrada "babxcc", que se han leído ya los símbolos subrayados en "babxcc", y la gramática es:

- A → a B c | x C | B
- B → bA
- C → c

- ¿Qué producción debe tomar para seguir el análisis?

- A → a B c | x C | B
- B → bA
- C → c

babxcc

La cadena de derivaciones ha sido:

- A → B → bA → baBc → babAc

- Ahora hay que seguir desarrollando la variable A utilizando los conjuntos de predicción.
- Como la siguiente letra es una .x. se elige la segunda opción (A → x C)

La gramática

- A → a B c | a C | B

No cumple los requisitos para LL(1) porque si aparece una "a" en la entrada hay dos posibles opciones.

Luego el análisis:

- no puede ser predictivo y,
- la gramática no es LL(1).
- Cálculo de los conjuntos de predicción
  - Cálculo de los primeros
  - Cálculo de los siguientes

Cálculo de los Conjuntos de Predicción

Los conjuntos de predicción se calculan:

- en función de los primeros símbolos que puede generar la parte derecha de la regla, y
- cuando la parte derecha puede generar la cadena vacía, en función de los símbolos que pueden aparecer a continuación de la parte izquierda de la regla en una forma sentencial derivable del símbolo inicial.

Para poder definir el conjunto de predicción es necesario determinar:

- ***conjunto de primeros***
  - calcular los primeros símbolos que genera
  - una cadena de terminales y no terminales
- ***conjunto de siguientes***
  - obtener los símbolos que pueden seguir a un no terminal en una forma sentencial.

Cálculo de los primeros

Sea una gramática $G = (\sum_T, \sum_N, S, P)$

– PRIM se aplica a $(\alpha \in (\sum_T \cup \sum_N)^*)$

Def.-

- Si α es una forma sentencial compuesta por una concatenación de símbolos, PRIM() es el conjunto de terminales (o λ) que pueden aparecer iniciando las cadenas que pueden derivar de α.
- Def. formal-

– $a \in \text{PRIM}(\alpha)$ si $a \in (\sum_T \cup \{\lambda\})$ y $\alpha \overset{*}{\Rightarrow} a\beta$

Reglas:

a) Si α coincide con λ, PRIM(λ)={λ}
b) Si $\alpha \in (\sum_T \cup \sum_N)^+$, $\alpha = a_1 a_2 \ldots a_n$, entonces
   a) Si $a_1 \equiv a \in \sum_T$, PRIM(α)={a}
   b) Si $a_1 \equiv A \in \sum_N$ es necesario obtener los primeros de todas las partes derechas de las producciones de A.

      $\text{PRIM(A)} = \cup_{i=1}^{n} \text{PRIM}(\alpha_i) \quad \alpha_i \in P$

      Si después de calcular PRIM(A), λ ∈ PRIM(A) y A no es el último símbolo de α, entonces

      $\text{PRIM}(\alpha) = (\text{PRIM(A)}-\{\lambda\}) \cup \text{PRIM}(a_2 \ldots a_n)$

      Tanto si A es el último símbolo de α, como si λ ∉ PRIM(A)

      $\text{PRIM}(\alpha) = \text{PRIM(A)}$

Gramática

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \lambda$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \lambda$
$F \rightarrow (E) \mid ident$

$PRIM(E')=\{+,\lambda\}$
$PRIM(T')=\{*,\lambda\}$
$PRIM(F)=\{(,ident\}$
$PRIM(E)=PRIM(T)=PRIM(F)=\{(,ident\}$

$E \rightarrow T\ E'$ $\quad$ $T \rightarrow F\ T'$

Cálculo de los Siguientes

Se aplica a no terminales ($\sum_N$) de la gramática (A)

- Devuelve el conjunto de terminales que pueden aparecer a continuación de A en alguna forma sentencial derivada del símbolo inicial y
- Un símbolo ($) que representa el final de la cadena de entrada.

Def.-

Si A es un símbolo inicial no terminal de la gramática, SIG(A) es el conjunto de terminales (y $) que pueden aparecer a continuación de A en alguna forma sentencial derivada del símbolo inicial.

Def. formal

$$a \in SIG(A) \text{ si } a \in (\sum_N \cup \{\$\}) \text{ y } \exists\ \alpha,\beta\ /$$
$$S \overset{*}{\Rightarrow} \alpha A a \beta \text{ para algún par de cadenas } \alpha$$

Reglas para el Cálculo del conjunto de los siguientes

1. Inicialmente, SIG(A) = ø
2. Si A es símbolo inicial, SIG(A) = SIG(A) ∪ {$}
3. Para cada regla B → αAβ
   SIG(A) = SIG(A) ∪ (PRIM(β) - {λ})
4. Para cada regla B → αA o B → αAβ en la que λ ∈ PRIM(β)
   SIG(A) = SIG(A) ∪ SIG(B)
5. Repetir los pasos 3 y 4 hasta que no se puedan añadir más símbolos a SIG(A)

Cálculo de los siguientes de la gramática:

E → T E'
E' → + T E' | λ
T → F T'
T' → * F T' | λ
F → (E) | ident

Gramática

SIG(E)

E → T E'
E' → + T E' | λ
T → F T'
T' → * F T' | λ
F → (E) | ident

SIG(E) = SIG(E) ∪ {$}
como F → (E)
SIG(E) = {$}∪ (PRIM()) - {λ})
SIG(E) = {$}∪ {)} = {$,)}

F → (E)
B → αAβ

Para cada regla B → αAβ
SIG(A) = SIG(A) ∪ (PRIM(β) - {λ})

Gramática

E → T E'
E' → + T E' | λ
T → F T'
T' → * F T' | λ
F → (E) | ident

SIG(E')

E → T E'
B → α A
E' → + T E'
B → α A

$SIG(E') = SIG(E') \cup SIG(E)$
$SIG(E') = SIG(E') \cup SIG(E')$
$SIG(E') = \emptyset \cup SIG(E) = \{\$,)\}$

Para cada regla B → αA o B → αAβ en la que λ ∈ PRIM(β)
SIG(A) = SIG(A) ∪ SIG(B)

Gramática

E → T E'
E' → + T E' | λ
T → F T'
T' → * F T' | λ
F → (E) | ident

SIG(T)

Para cada regla B → αAβ
SIG(A) = SIG(A) ∪ (PRIM(β) - {λ})
Para cada regla B → αA o B → αAβ
en la que λ ∈ PRIM(β)
SIG(A) = SIG(A) ∪ SIG(B)

PRIM(E') = { + , λ }

$SIG(T) = SIG(T) \cup (PRIM(E')-\{\lambda\}) = \{+\}$

como λ ∈ PRIM(E')

$SIG(T) = SIG(T) \cup SIG(E') = \{\$,)\}$

como E' → λ

$SIG(T) = SIG(T) \cup SIG(E')$

$SIG(T) = (\{+\}) \cup \{\$,)\} = \{+ , \$,)\}$

E → T E'
B → α A β
E' → + T E'
B → α A β

Gramática

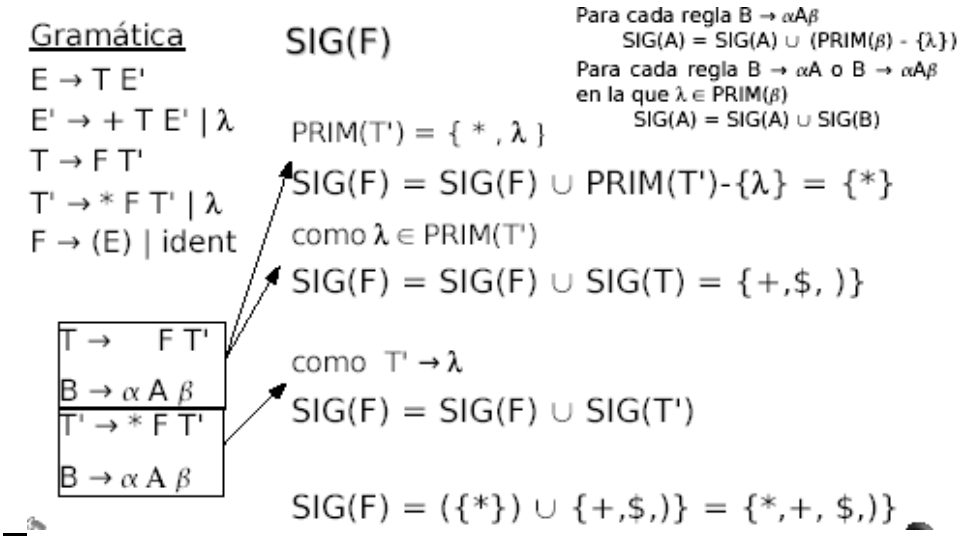
E → T E'
E' → + T E' | λ
T → F T'
T' → * F T' | λ
F → (E) | ident

SIG(T')

$SIG(T') = SIG(T') \cup SIG(T)$
$SIG(T') = SIG(T') \cup SIG(T')$
$SIG(T') = \emptyset \cup SIG(T) = \{+ , \$,)\}$

T → F T'
B → α A
T' → * F T'
B → α A

Para cada regla B → αA o B → αAβ en la que λ ∈ PRIM(β)
SIG(A) = SIG(A) ∪ SIG(B)



Cálculo de los conjuntos de predicción

La función PRED

- se aplica a producciones de la gramática

$$(A \rightarrow \alpha)$$

- devuelve un conjunto de predicción que puede contener cualesquiera de los terminales de la gramática y el símbolo $, pero nunca puede contener λ.

Cuando el ASDP tiene que derivar un no Terminal

- consulta el símbolo de entrada y lo busca en los conjuntos de predicción de cada regla de ese no terminal.
- si los conjuntos de predicción son disjuntos, el AS podrá construir una derivación por la izda. de la cadena de entrada.

PRED (A → α) =
si λ ∈ PRIM(α) entonces = (PRIM(α) - {λ}) ∪ SIG(A)
si no = PRIM(α)

$S \rightarrow A\ B \mid s$

$A \rightarrow a\ S\ c \mid e\ B\ f \mid \lambda$

$B \rightarrow b\ A\ d \mid \lambda$

PRIM(AB) = {a , e , b, λ}
SIG(S) A → a S c
SIG(S) = SIG(S) ∪ (PRIM(c) - {λ}) = {$, c}

. . . . . .

PRED(S → AB) = (PRIM(AB)-{λ}) ∪ SIG(S) = {a,e,b,$,c}
PRED(S → s) = PRIM(s) = {s}
PRED(A → aSc) = PRIM(aSc) = {a}
PRED(A → eBf) = PRIM(eBf) = {e}
PRED(B → bAd) = PRIM(bAd) = {b}

$S \rightarrow A\ B \mid s$

$A \rightarrow a\ S\ c \mid e\ B\ f \mid \lambda$

$B \rightarrow b\ A\ d \mid \lambda$

$PRED(A \rightarrow \lambda) = (PRIM(\lambda)-\{\lambda\}) \cup SIG(A)$

SIG(A) = SIG(A) ∪ (PRIM(B) - {λ}) = {b , λ} - {λ}={b}

PRIM(B) = { b , λ } es decir, λ ∈ PRIM(B)

$S \rightarrow A\ B$
$B \rightarrow \alpha\ A\ \beta$
$B \rightarrow b\ A\ d$
$B \rightarrow \alpha\ A\ \beta$

SIG(A) = SIG(A) ∪ SIG(S) = {$, c}
SIG(A) = SIG(A) ∪ (PRIM(d) - {λ}) = {d}
SIG(A) = {b, $, c , d}

$PRED(A \rightarrow \lambda) = \{b, \$, c , d\}$

$PRED(B \rightarrow \lambda) = (PRIM(\lambda)-\{\lambda\}) \cup SIG(B)$

$S \rightarrow A\ B \mid s$

$A \rightarrow a\ S\ c \mid e\ B\ f \mid \lambda$

$B \rightarrow b\ A\ d \mid \lambda$

$S \rightarrow A\ B$
$B \rightarrow \alpha\ A$
$A \rightarrow e\ B\ f$
$B \rightarrow \alpha\ A\ \beta$

SIG(B) = SIG(B) ∪ SIG(S) = {$, c}

SIG(B) = SIG(B) ∪ (PRIM(f) - {λ}) = {f}

SIG(B) = {$, c} ∪ {f} = {$, c , f}

$PRED(B \rightarrow \lambda) = \{\$, c , f\}$

La Condición LL(1)

Gramáticas LL(1), se debe cumplir:

- Dadas las producciones de la gramática para un mismo terminal $A \rightarrow \alpha_1 | \alpha_2 | ... | \alpha_n$, $\forall A \in \sum_N$, se debe cumplir la condición:

  $\forall$ i,j (i≠j) $PRED(A \rightarrow \alpha_i) \cap PRED(A \rightarrow \alpha_j) = \emptyset$

  - Es decir, no puede haber ningún símbolo terminal que pertenezca a dos o más conjuntos de predicción de las reglas de un mismo no terminal.

Características de la condición LL(1)

- La secuencia de *tokens* se analiza de izquierda a derecha.
- Siempre deriva el no terminal que aparezca más a la izquierda.
- Sólo es necesario ver un *token* de la secuencia de entrada para averiguar que regla de producción seguir.

Ejemplo de la Gramática LL(1)

Gramática	conjuntos de predicción
A → abB	{a}
A → Bb	{b,c}
B → b	{b}
B → c	{c}

Es LL(1) porque son disjuntos

$PRED(A \rightarrow abB) \cap PRED(A \rightarrow Bb) = \{a\} \cap \{b,c\} = \emptyset$

$PRED(B \rightarrow b) \cap PRED(B \rightarrow c) = \{b\} \cap \{c\} = \emptyset$

Si se añade la regla **B → a**

Gramática	conjuntos de predicción
A → abB	{a}
A → Bb	{a,b,c}
B → b	{b}
B → c	{c}
B → a	{a}

No es LL(1) porque no son disjuntos

$$PRED(A \rightarrow abB) \cap PRED(A \rightarrow Bb) = \{a\} \cap \{a,b,c\} \neq \emptyset$$

¿Cumple esta gramática la condición LL(1)?

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow num \mid (E)$$

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

$PRED(E \rightarrow E + T) = PRIM(E + T) = \{num,(\}$
$PRED(E \rightarrow T) = PRIM(T) = \{num,(\}$
$PRED(T \rightarrow T * F) = PRIM(T * F) = \{num,(\}$
$PRED(T \rightarrow F) = PRIM(F) = \{num,(\}$
$PRED(F \rightarrow num) = PRIM(num) = \{num\}$
$PRED(F \rightarrow (E)) = PRIM((E)) = \{(\}$

$PRED(E \rightarrow E + T) \cap PRED(E \rightarrow T) = \{num,(\} \cap \{num,(\} \neq \emptyset$
$PRED(T \rightarrow T * F) \cap PRED(T \rightarrow F) = \{num,(\} \cap \{num,(\} \neq \emptyset$
$PRED(F \rightarrow num) \cap PRED(F \rightarrow (E)) = \{num\} \cap \{(\} = \emptyset$

Los no terminales E y T no cumplen la condición LL(1)

Modificación de gramáticas no LL(1)

- Eliminación de la ambigüedad
- Factorización por la izquierda
- Eliminación de la recursividad por la izquierda

Características

- Algunas características garantizan que una gramática no es LL(1):
  - Recursiva por la izquierda
  - Símbolos comunes por la izquierda
  - Ambigua
- Existen métodos para modificarla y convertirla en una gramática LL(1)

Eliminación de la Ambigüedad

- Más de un árbol sintáctico posible.
- No existe una metodología para eliminarla
- ***Solución:*** replantearse el diseño de la misma para encontrar una gramática no ambigua equivalente (que genere el mismo lenguaje).
-

Factorización por la Izquierda:

- Si dos producciones alternativas de un símbolo A empiezan igual, no se sabrá por cuál de ellas seguir.
- ***Solución:*** reescribir las producciones de A para retrasar la decisión hasta haber visto lo suficiente de la entrada como para elegir la opción correcta.

Regla general para factorizar por la izquierda:

- Encontrar el prefijo más largo común a dos o más producciones de A, pero siempre aquél que sea común a más producciones
- Si existe un prefijo común más corto en varias producciones y otro más largo en un par de ellas, hay que eliminar primero el más corto común a las varias ($\gamma = \delta\beta$ tal que $|\delta| < |\gamma|$).

Solución: sustituir las producciones:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_i$$

Donde γ_i indica todas las alternativas que no empiezan por α, por:

$$A \rightarrow \alpha A' \mid \gamma_i$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

- Sea la gramática

  Sent → if *Expr* then *Sent* else *Sent* endif

  Sent → if *Expr* then *Sent* endif

  Sent → *otras*

Donde se tiene

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

- Solución: sustituirlo por dos producciones de la forma

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

Gramática

Sent → if *Expr* then *Sent* else *Sent* endif
Sent → if *Expr* then *Sent* endif
Sent → *otras*

$\alpha \equiv$ if *Expr* then *Sent*
$\beta_1 \equiv$ else *Sent* endif
$\beta_2 \equiv$ endif
$\gamma_i \equiv$ *Otras*

Sent → if *Expr* then *Sent* Sent'
Sent → *Otras*
Sent'→ else *Sent* endif
Sent'→ endif

Eliminación recursiva por la izquierda:

- Una gramática es recursiva por la izquierda.
  - si tiene alguna producción que sea recursiva por la izquierda, o bien
  - si a partir de una forma sentencial como $A\gamma$ se obtiene una forma sentencial $A\beta\gamma$ en la que el no terminal A vuelve a ser el primero por la izquierda.

  $\exists A \in N / A \overset{*}{\Rightarrow} A\alpha$ para alguna cadena α

Regla para modificar una gramática

- Regla para modificar una gramática y deje de ser recursiva por la izquierda.

  Para una gramática

  $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

  se sustituye por

  $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$ (nuevo no terminal auxiliar)

  $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \lambda$ (recursiva por la derecha)

  $A' \rightarrow \lambda$

Gramática

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow num \mid (E)$

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

se sustituye por

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \lambda$

$A' \rightarrow \lambda$

$\alpha_1 \equiv + T \qquad \alpha_1 \equiv * F$

$\alpha_2 \equiv - T \qquad \alpha_2 \equiv / F$

$\beta \equiv T \qquad \beta \equiv F$

$E \rightarrow TE'$

$E' \rightarrow + TE' \mid -TE' \mid \lambda$

$T \rightarrow FT'$

$T' \rightarrow * FT' \mid / FT' \mid \lambda$

$F \rightarrow num \mid (E)$

Ejemplo de una conversión de una gramática en LL(1)

$S \rightarrow S\ inst \mid T\ R\ V$

$T \rightarrow tipo \mid \lambda$

$R \rightarrow blq\ V\ fblq \mid \lambda$

$V \rightarrow id\ S\ fin \mid id\ ; \mid \lambda$

$S \rightarrow T\ R\ V\ S'$

$S' \rightarrow inst\ S' \mid \lambda$

$T \rightarrow tipo \mid \lambda$

$R \rightarrow blq\ V\ fblq \mid \lambda$

$V \rightarrow id\ V' \mid \lambda$

$V' \rightarrow S\ fin \mid ;$

Recursiva por la izda. Y las dos primeras de V tienen factores comunes por la izda.

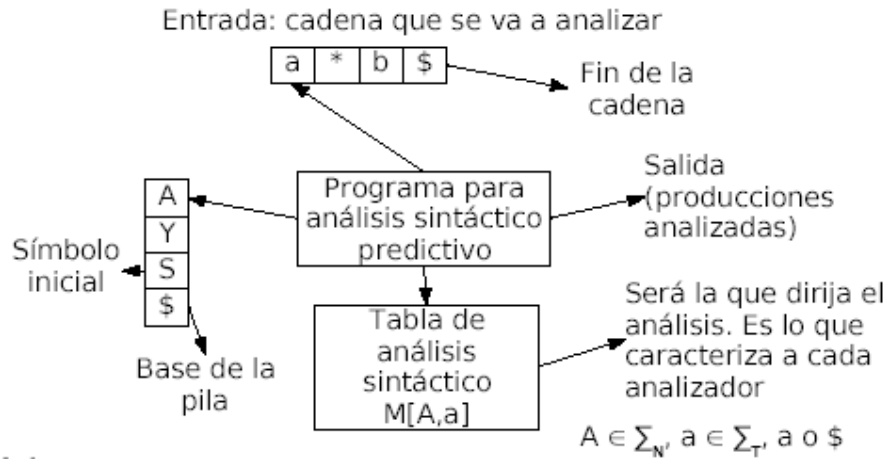
PRED(S' → inst S') = {inst}
PRED(S' → λ) = {$, fin}
PRED(T → tipo) = {tipo}
PRED(T → λ) = {blq, id, inst, $, fin}
PRED(R → blq V fblq) = {blq}
PRED(R → λ) = {id, inst, $, fin}
PRED(V → id V') = {id}
PRED(V → λ) = {inst, $, fin}
PRED(V' → S fin) = {tipo, blq, id, inst, $, fin}
PRED(V' → ;) = {;}

Analizador sintáctico descendente predictivo dirigido por tabla

- Modelo de un ASDP no recursivo dirigido por tabla
- Modelo del analizador sintáctico predictivo
- Construcción de la tablas de análisis sintáctico
- Procedimiento para construir tablas de análisis LL(1)
- Mensajes de error de tipo sintáctico

Características

- Es otra forma de construir un ASDP
- Construcción utilizando una pila de símbolos (terminales y no terminales)
- A la vista de un *token* de preanálisis se buscará en la tabla de análisis.
- Primero construir la tabla y después realizar el proceso de análisis.



Explicación del gráfico

- El programa tiene en cuenta A, el símbolo de la cima de la pila, y*a*, el símbolo en curso de la entrada.

A, el símbolo de la cima de la pila — ***a***, el símbolo en curso de la entrada.

Posibilidades:

1. Si A = *a* = $, el AS se detiene y anuncia el éxito de la realización del análisis.
2. Si A = *a* ≠ $, el analizador sintáctico saca A de la pila y mueve el apuntador al siguiente símbolo de la pila.
3. Si A es un no terminal, el programa consulta la entrada M[A,*a*] de la tabla.
   Si M[A,*a*]={A → UVW}, el AS sustituye A por WVU (U en la cima);
   Si M[A,*a*]=error, el AS llama a la rutina de recuperación error

Construcción de las tablas de análisis Sintáctico

- Algoritmo

  A → α con $a \in$ PRIM(α)

  El AS expandirá A por α cuando el símbolo de la entrada sea *a*.

  Problema

  cuando $\alpha = \lambda$ o $\alpha \overset{*}{\Rightarrow} a$

  en este caso se debe expandir A en α

  - si el símbolo de la entrada está en SIG(A), o
  - si se ha alcanzado $ y está en SIG(A)

- Método
  - realizar para cada producción A → α de G
    - para cada $a \in$ T, $a \in$ PRIM(α) se añade

      A → α a M[A,*a*]
  - Si $\lambda \in$ PRIM(α) se añade

    A → α a M[A,*b*] $\forall b \in$ SIG(A)
  - Si $\lambda \in$ PRIM(α) y $ $\in$ SIG(A) se añade

    A → α a M[A,$]
  - Cada entrada no definida se marca con *error* en M

Procedimientos para construir tablas de análisis

- Las filas se etiquetan con los no terminales y las columnas se etiquetan con los terminales y el símbolo fin de fichero.
  - En cada celda se colocará la regla a aplicar para analizar la variable de esa fila cuando el terminal de esa columna aparezca en la entrada.
- Se calculan los conjuntos de predicción para cada regla.
- Para cada terminal $a \in PRED(A \rightarrow \alpha)$, se añade $A \rightarrow \alpha$ a Tabla[A, a]
- Cada entrada no definida de Tabla será error sintáctico.

Gramática
E → TE'
E'→ + TE' | -TE' | λ
T → FT'
T' → * FT' | / FT' | λ
F → num | (E)

PRED(E → TE') = PRIM(T) =
= PRIM(F) = {num,(}
PRED(E' → +TE') = {+}
PRED(E' → -TE') = {-}
PRED(E' → λ) = SIG(E') =
= SIG(E) = {),$}
PRED(T → FT') = PRIM(F) = {(,num}
PRED(T' → *FT') = {*}
PRED(T' → /FT') = {/}
PRED(T' → λ) = {+, -,), $}
PRED(F → (E)) = {(}
PRED(F → num) = {num}

	num	+	-	*	/	(	)	$
E	E → TE'	Error	Error	Error	Error	E → TE'	Error	Error
E'	Error	E' → +TE'	E' → -TE'	Error	Error	Error	E' → λ	E' → λ
T	T → FT'	Error	Error	Error	Error	T → FT'	Error	Error
T'	Error	T' → λ	T' → λ	T' → *FT'	T' → /FT'	Error	T' → λ	T' → λ
F	F → num	Error	Error	Error	Error	F → (E)	Error	Error

Analizador que, al ser predictivo, sólo podrá construirse si la gramática a analizar es LL(1).

Será LL(1) porque no aparerán dos o más producciones en la misma casilla.

Gramática

Sent → if *Expr* then *Sent* *Sent'*
Sent → *otras*
Sent' → *else Sent* | λ
Expr → logico

	otras	*lógico*	*else*	*if*	*then*	$
Sent	Sent → otras	Error	Error	Sent → if ...	Error	Error
Sent'	Error	Error	Sent' → else Sent Sent' → λ	Error	Error	Sent' → λ
Expr	Error	Expr → lógico	Error	Error	Error	Error

Gramática

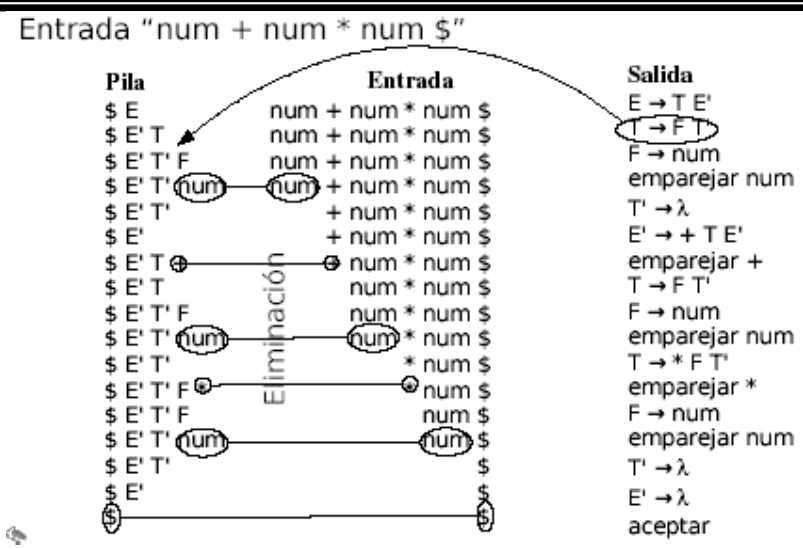
E → TE'
E'→ + TE' | -TE' | λ
T → FT'
T' → * FT' | / FT' | λ
F → num | (E)

Tabla de análisis

	num	+	–	*	/	(	)	$
E	E → TE'	Error	Error	Error	Error	E → TE'	Error	Error
E'	Error	E' → +TE'	E' → -TE'	Error	Error	Error	E' → λ	E' → λ
T	T → FT'	Error	Error	Error	Error	T → FT'	Error	Error
T'	Error	T' → λ	T' → λ	T' → *FT'	T' → /FT'	Error	T' → λ	T' → λ
F	F → num	Error	Error	Error	Error	F → (E)	Error	Error

Aplicar el algoritmo a la entrada num + num * num $

- Pila
  - muestra el contenido de la pila en cada momento
    fondo -------- tope
- Entrada
  - representa lo que queda por analizar en cada momento.
- Salida
  - emparejamiento de *tokens*
  - indica las producciones que se van aplicando


Entrada "num + num * num $"
Pila
Entrada
Salida
$ E
num + num * num $
E → T E'
$ E' T
num + num * num $
T → F T'
$ E' T' F
num + num * num $
F → num
$ E' T' num
num + num * num $
emparejar num
$ E' T'
+ num * num $
T' → λ
$ E'
+ num * num $
E' → + T E'
$ E' T
num * num $
emparejar +
$ E' T
num * num $
T → F T'
$ E' T' F
num * num $
F → num
$ E' T' num
num * num $
emparejar num
$ E' T'
* num $
T → * F T'
$ E' T' F
num $
emparejar *
$ E' T' F
num $
F → num
$ E' T' num
num $
emparejar num
$ E' T'
$
T' → λ
$ E'
$
E' → λ
$
$
aceptar
Eliminación

Análisis Sintáctico 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 Є 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 Є (N U T)*

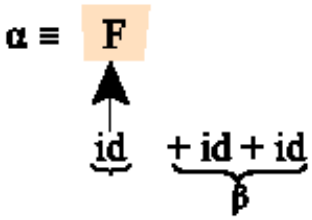
Por ejemplo, si quisiéramos reconocer "id + id + id", partiendo de la gramática del cuadro

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

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):

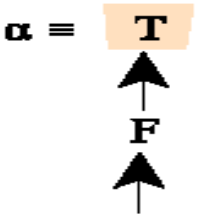
y tras consumir el primer *token*:

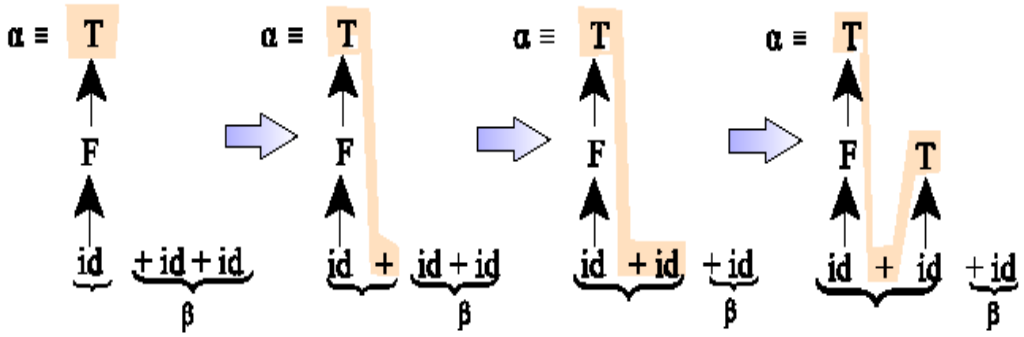
y ahora podemos aplicar la regla 5 hacia atrás, con lo que α es F:



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

Así, poco a poco se va construyendo el árbol:





Como puede verse, á representa al árbol sintáctico visto desde arriba conforme se va construyendo ascendentemente: α es una forma sentencial 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 á-â) 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 otro 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:

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

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

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

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

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

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

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

puede reducirse por una regla de la forma:

$$A_k \rightarrow \varepsilon$$

obteniéndose:

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

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:

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

un desplazamiento pasaría a:

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

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 en medio. 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.

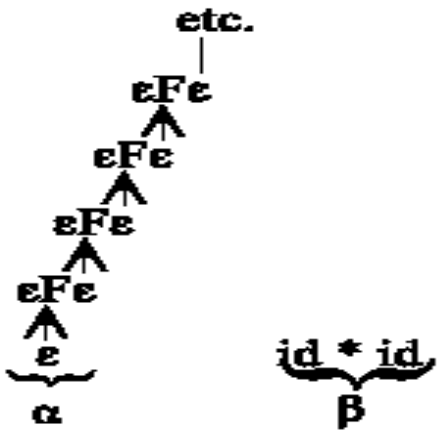
Análisis ascendente con retroceso

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.

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

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

Nota: Nótese como la cola de á puede ser ε, εg, 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.

Análisis ascendente de gramáticas LR(1)

En esta parte 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.

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 "***reemplazo***" 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.

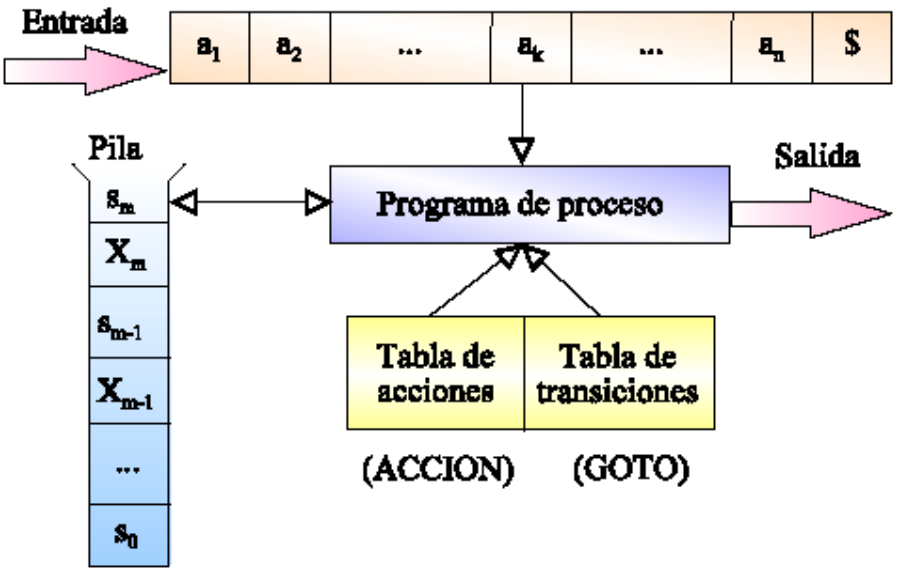
Funcionalmente hablando, un analizador LR consta de dos partes bien diferenciadas:

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 siguiente 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)).



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. Cada uno de los Xi es un símbolo de la gramática (Xi Є (N U T) *) y los s representan distintos estados del autómata asociado a la gramática; al conjunto de estados lo denominaremos E: S_i Є E. Cada estado s tiene asociado un símbolo X , 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 transitar a otro estado (GOTO) y la tarea de qué acción realizar (ACCION). La tabla GOTO es de la forma E × (N U T U {$}) y contiene estado de E, mientras que la tabla ACCION es de la forma E × (N U T) y contiene una de las cuatro acciones que vimos en el apartado anterior.

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 , a) en la tabla de ACCION. El contenido de la casilla puede ser:

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

   Si se acepta o rechaza, se da por finalizado el análisis, si se desplaza se mete a_j en la pila, y si se reduce, entonces la cima de pila coincide con el consecuente λ (amén de los estados), y se sustituye λ 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 ACCION(s_m , a_j) = Desplazar, entonces se introduce en la pila el símbolo a_i , produciendo:

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

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

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

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

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

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

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

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 á 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 siguiente. 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

- D-i: significa “desplazar y pasar al estado i” (ACCION y GOTO todo en uno, 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 anterior 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+ 0 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
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 á 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.

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.