Modelos básicos de Computación. Máquinas de Turing. Autómatas Finitos. Autómatas de Pila.

Tema 5(II). Autómatas con Pila(I). Analizadores LL(K).

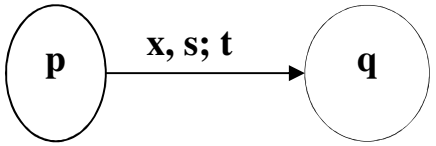
Algorítmica y Modelos de Computación

Índice

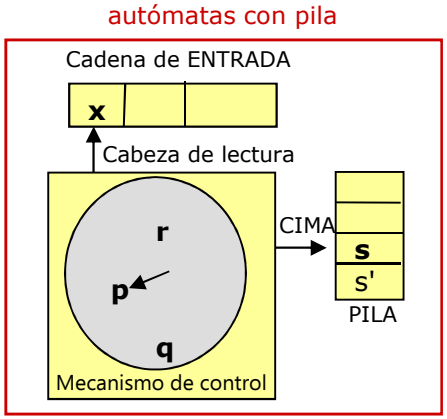
Índice. ANÁLISIS SINTÁCTICO DESCENDENTE

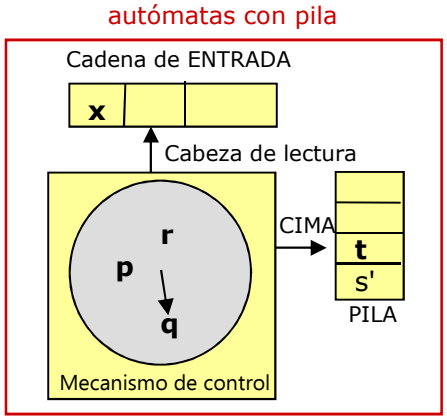
3.1. Introducción

- Los ***Autómatas con Pila*** son una extensión de los AFD a los que se les añade una memoria (pila).
- En la pila se almacenan símbolos de la cadena de entrada y de la gramática, así como caracteres especiales (#) para indicar el estado de pila vacía.
- Las transiciones son de la forma: **(p,x,s;q,t)**
  - p=estado inicial
  - q=estado al que llega
  - x= símbolo de la cadena de entrada
  - s=símbolo que se desapila
  - t=símbolo que se apila



- Gráficamente:





3.2. Definición de Autómatas con pila.

- **Definición:** Un Autómata a Pila se define como la séptupla $(\sum, \Gamma, \mathbf{Q}, \mathbf{A0}, \mathbf{q0}, \mathbf{f}, \mathbf{F})$
  - $\sum$: alfabeto de entrada.
  - Γ : alfabeto de la pila.
  - Q: conjunto de estados.
  - A_0: símbolo inicial de la pila (#).
  - q_0: símbolo inicial del conjunto de estados.
  - f: función de transición. Es una aplicación de

$$f\colon Q\ x\ \sum \cup \{\lambda\}\ x\ \Gamma \rightarrow P(Q\ x\ \Gamma^*)$$

  - F: conjunto de estados finales o de aceptación.

- **Función de transición.** Interpretamos la función *f* de la siguiente forma:

➢ $\boldsymbol{f(q, a, A) = \{(q_1, Z_1), \ldots, (q_n, Zn)\}}$

Si el AP se encuentra en el estado $\boldsymbol{q}$, lee el símbolo *a* de la cinta de entrada, y aparece el símbolo *A* en el tope de la pila, pasará al estado $\boldsymbol{q_i}$ **(n≥ i≥ 1),** borrará el símbolo ***A*** de la pila e introducirá la palabra $\boldsymbol{Z_i}$, situando la cabecera de la misma en el tope de la pila, y avanzando una posición en la cinta de entrada.

➢ $\boldsymbol{f(q, \lambda, A) = \{(q_1, Z_1), \ldots, (q_n, Zn)\}}$

Si el AP se encuentra en el estado *q* y aparece el símbolo *A* en el tope de la pila, pasará al estado $\boldsymbol{q_i}$ $(\boldsymbol{n \geq i \geq 1})$**,** borrará el símbolo *A* de la pila e introducirá la palabra $\boldsymbol{Z_i}$, situando la cabecera de la misma en el tope de la pila, y mantendrá la misma posición en la cinta de entrada.

3.2. Definición de Autómatas con pila.

Descripciones instantáneas

Dado un AP, podemos describir el proceso de aceptación o rechazo de una palabra de $\sum^*$ mediante una serie de descripciones instantáneas de la forma $(\boldsymbol{q}, \boldsymbol{x}, \boldsymbol{Z})$ que definen respectivamente el estado del AP, la entrada que queda por leer, y el contenido de la pila en un momento dado.

- Decimos que una descripción instantánea (q, az, AZ) **precede** a otra (p, z, YZ) en ***un paso*** y se expresa como $(\boldsymbol{q}, \boldsymbol{az}, \boldsymbol{AZ}) \rightarrow (\boldsymbol{p}, \boldsymbol{z}, \boldsymbol{YZ})$, $si\ (p, Y) \in f(q, a, A)$.
- Decimos que una descripción instantánea (q, az, AZ) **precede** a otra (p, z, YZ) en $\boldsymbol{n}$ ***pasos*** y se expresa como $(\boldsymbol{q}, \boldsymbol{az}, \boldsymbol{AZ}) \rightarrow^* (\boldsymbol{p}, \boldsymbol{z}, \boldsymbol{YZ})$, si existen una serie de descripciones que cumplen la relación de precedencia anterior de una en una.

- **Autómata a Pila Determinista:** Un AP es Determinista si verifica:

1. $\forall q \in Q, \forall A \in \Gamma, cardinal(f(q, \lambda, A)) > 0 \Rightarrow f(q, a, A) = \varnothing, \forall a \in \sum.$
2. $\forall q \in Q, \forall A \in \Gamma, \forall a \in \sum \cup \{\lambda\} \Rightarrow cardinal(f(q, a, A)) < 2.$

3.3. Lenguaje aceptado por un autómata con pila.

- Lenguaje aceptado por un AP $M = (\sum, \Gamma, Q, A_0, q_0, f, F)$
  - **Lenguaje aceptado por criterio de estado final**

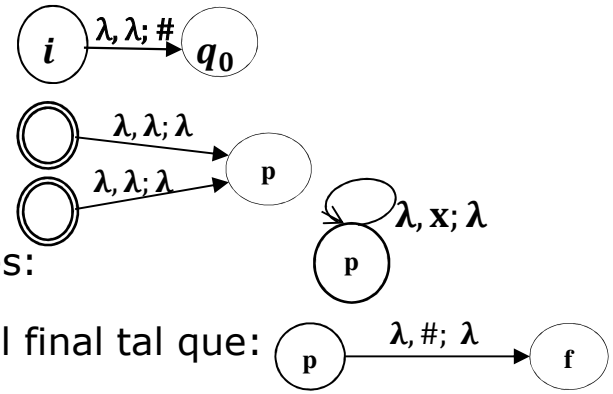
$$LEF(M) = \{x \,/\, (q_0, x, A_0) \rightarrow * \ (p, \lambda, X), p \in F, X \in \Gamma^*\}$$

  - **Lenguaje aceptado por criterio de pila vacía**

$$LPV(M) = \{x \,/\, (q_0, x, A_0) \rightarrow * \ (p, \lambda, \lambda), p \in Q\}$$

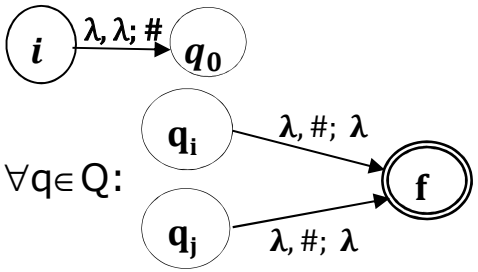
- **Equivalencia**
  - $LEF \Rightarrow LPV$

1. Nuevo estado i tal que:
2. Nuevo estado p tal que:
3. $\forall x \in \Gamma$ crear las transiciones:
4. Nuevo estado f que será el final tal que:



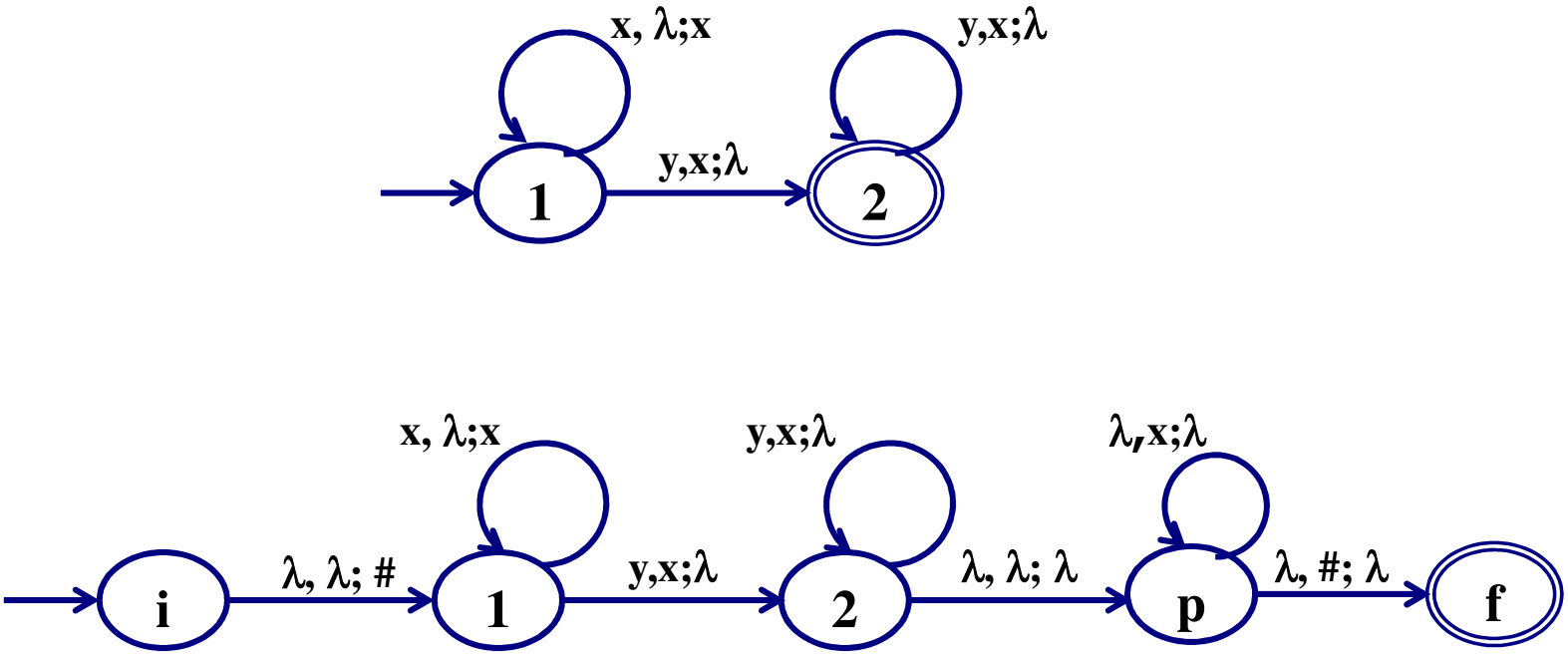
  - LPV $\Rightarrow$ LEF

1. Igual que el anterior.
2. Nuevo estado f tal que $\forall q \in Q$:



3.3. Lenguaje aceptado por un autómata con pila. Ejemplo.

- **Ejemplo**: Autómata con pila que reconoce el lenguaje $L = \{x^n y^m, n > m\}$
  - Aplicamos el algoritmo de $\boldsymbol{LEF} \Rightarrow \boldsymbol{LPV}$



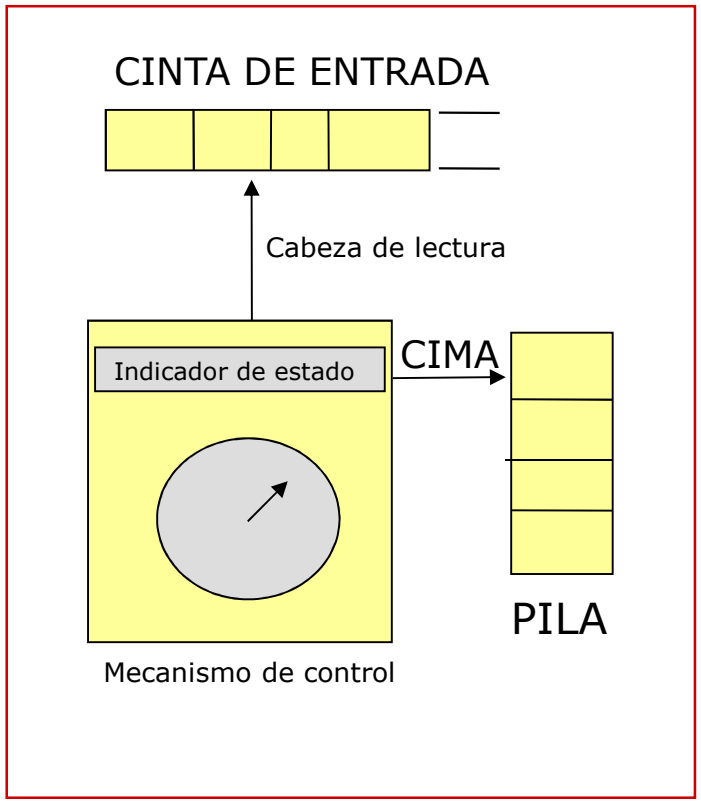
3.3. Lenguaje aceptado por un autómata con pila.

- Los *lenguajes libres de contexto* son reconocidos por los *Autómatas a Pila No Determinísticos*.
- Los *APND* aceptan más lenguajes que los Autómatas a Pila Determinísticos, por lo que **no** son *equivalentes*, aunque la mayoría de los lenguajes de programación son reconocidos por los *APD.*
- **Teorema**
  - A toda gramática G libre de contexto le corresponde biunívocamente un AP que acepta por vaciado de pila el lenguaje generado por ella.
- Existen dos métodos para construir un **AP a partir de la GIC**:
  - Análisis descendente: gramáticas **LL(k)**
  - Análisis ascendente: gramáticas **LR(k)**

    donde:
    - **L** ⇒ ***Left to right***: la secuencia de tokens de entrada se analiza de izquierda a derecha.
    - **L/R** ⇒ ***Left-most/right-most***: obtiene la derivación por la izquierda/derecha.
    - **k** ⇒ es el número de símbolos de entrada que es necesario conocer en cada momento para poder hacer el análisis.

3.4.1.1. Proceso de Análisis Sintáctico Descendente.

autómatas con pila



- El proceso comienza apilando el axioma de la gramática. Continuamente mira lo que hay en las últimas casillas de la pila:
  - Si existe concordancia con un no terminal de alguna de las reglas de producción de la gramática analizada, elimina de la cima de la pila ese no terminal y lo cambia por la cadena de la parte derecha de la regla de producción.
  - Si existe concordancia con un terminal y coincide con el carácter de la entrada, lo desapila y lee un carácter más de la entrada.
  - Si con este proceso se consigue agotar el contenido de la cinta de entrada y de la pila, la palabra es reconocida. En otro caso no lo es.

3.4.1.1. Proceso de Análisis Sintáctico.

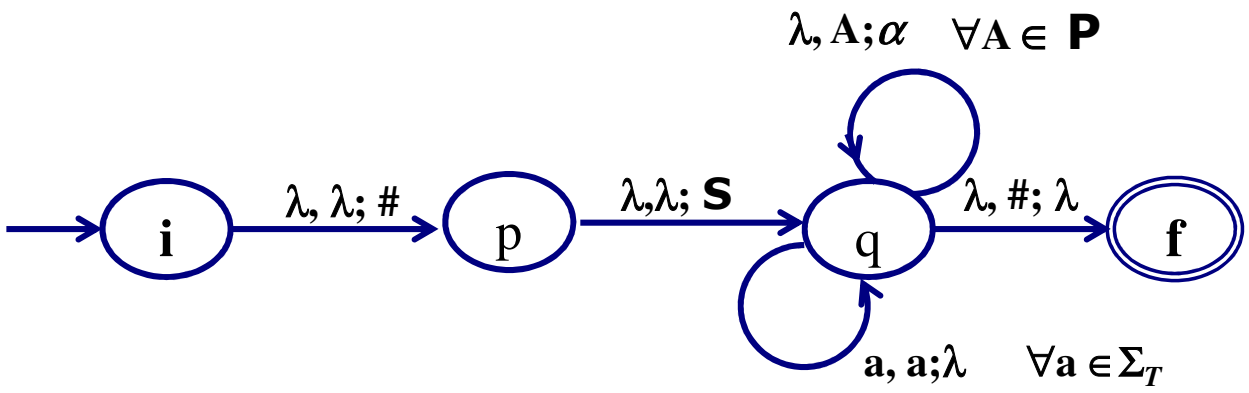
Ejemplo: G = ({a}, {S}, S, {S ::= aS | a})

cadena de entrada= aa

PILA	ENTRADA	ACCIÓN
λ	*aa$*	Apilar S
S	*aa$*	*S::=aS*
aS	*aa$*	*Desapilar a; Leer()*
S	*a$*	*S::=a*
a	*a$*	*Desapilar a; Leer()*
λ	*$*	*aceptar*

3.4.1.2. Analizadores LL y autómatas de pila no deterministas

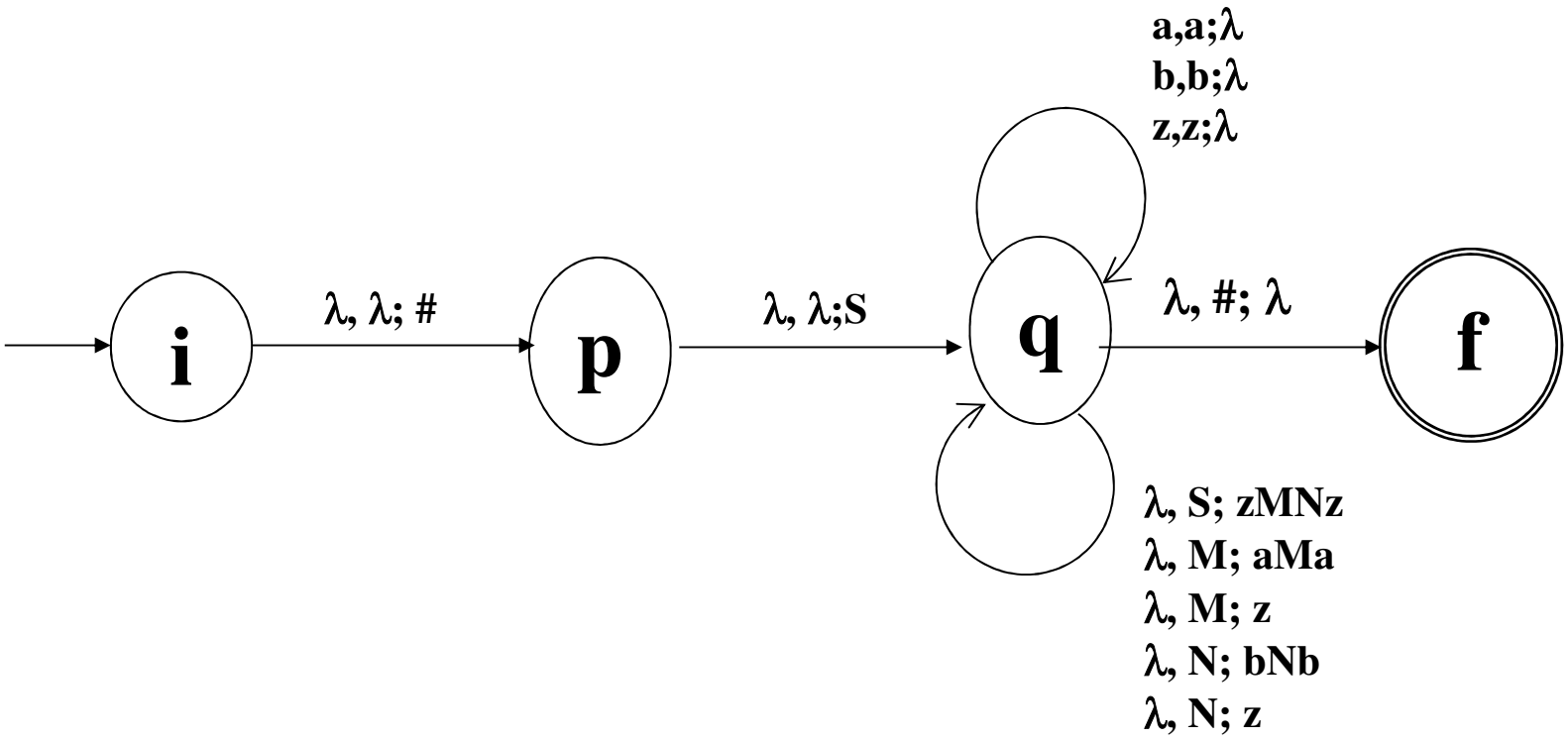
1. Construir los cuatro estados del autómata: *i* el inicial, *f* el de aceptación y *p* y *q* dos estados intermedios.
2. Se introducen las transiciones estándar para marcar la cima de la pila, para apilar el axioma y para determinar cuándo queda vacía:
   $(i,\lambda,\lambda;\ p,\#)$, $(p,\lambda,\lambda;\ q,S)$ y $(q,\lambda,\#;\ f,\lambda)$.
3. Por cada símbolo terminal a $\in \Sigma_T$, introducimos una transición de la forma $(q,a,a;q,\lambda)$.
4. Por cada regla de la forma **$A ::= \alpha \in P$,** añadimos la transición $(q,\lambda,A;q,\alpha)$.



3.4.1.2. Analizadores LL y autómatas de pila no deterministas

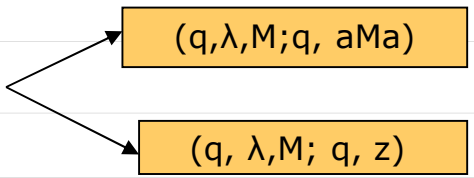
- **Ejemplo**:

G = ({a, b, z}, {S, M, N}, S, {S ::= zMNz, M ::= aMa | z, N ::= bNb | z})



3.4.1.2. Analizadores LL y autómatas de pila no deterministas

estado	Pila	Entrada	Acción	indeterminación
i	λ	zazabzbz$	(i, λ, λ ; p, #)	
p	#	zazabzbz$	(p, λ, λ ; q, S)	
q	S#	zazabzbz$	(q, λ, S ; q, zMNz)	
q	zMNz#	zazabzbz$	(q, z,z;q,λ)	
q	MNz#	azabzbz$	**(q,λ,M;q, aMa)** | (q, λ, M ; q, z)	
q	aMaNz#	azabzbz$	(q, a,a;q,λ)	
q	MaNz#	zabzbz$	**(q, λ, M ; q, z)**| (q,λ,M;q, aMa)	
q	zaNz#	zabzbz$	(q, z,z;q,λ)	
q	aNz#	abzbz$	(q, a,a;q,λ)	
q	Nz#	bzbz$	**(q,λ,N;q, bNb)** | (q, λ, N ; q, z)	
q	bNbz#	bzbz$	(q, b,b;q,λ)	
q	Nbz#	zbz$	(q, λ, N ; q, z)	
q	zbz#	zbz$	(q, z,z;q,λ)	
q	bz#	bz$	(q, b,b;q,λ)	
q	z#	z$	(q, z,z;q,λ)	
q	#	$	(q, λ, #; f, λ)	
f	λ	λ	Aceptar	

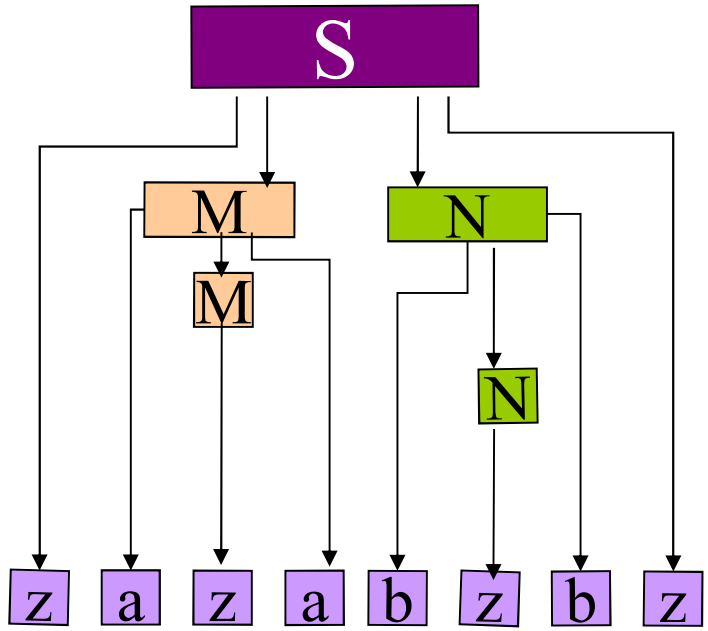


3.4.1.2. Analizadores LL y autómatas de pila no deterministas

Acción
(i, λ, λ ; p, #)
(p, λ, λ ; q, S)
(q, λ, S ; q, zMNz)
(q, z,z;q,λ)
(q,λ,M;q, aMa) | (q, λ, M ; q, z)
(q, a,a;q,λ)
(q, λ, M ; q, z)| (q,λ,M;q, aMa)
(q, z,z;q,λ)
(q, a,a;q,λ)
(q,λ,N;q, bNb) | (q, λ, N ; q, z)
(q, b,b;q,λ)
(q, λ, N ; q, z)
(q, z,z;q,λ)
(q, b,b;q,λ)
(q, z,z;q,λ)
(q, λ, #; f, λ)

Aceptar

Si aceptamos una cadena usando este autómata podremos reconstruir la secuencia de derivaciones que nos conduce a ella a partir de *S.*



$$S \rightarrow zMNz \rightarrow zaMaNz \rightarrow zazaNz \rightarrow zazabNbz \rightarrow zazabzbz$$

3.4.1.3. Implementación de Analizadores LL

- Los analizadores sintácticos descendentes son lo que construyen el árbol sintáctico de la sentencia a reconocer de una forma descendente, comenzando por el símbolo inicial o raíz, hasta llegar a los símbolos terminales que forman la sentencia.
- Se desea un análisis sintáctico descendente sin retroceso, por medio del uso de gramáticas LL(k=1):
  - pueden analizar sintácticamente sin retroceso, en forma descendente, examinando en cada paso todos los símbolos procesados anteriormente y los *k=1* símbolos más a la derecha
- Para eliminar el retroceso en el análisis descendente, se ha de elegir correctamente la producción correspondiente a cada no terminal que se expande. Es decir que el análisis descendente ha de ser **determinista**, y sólo se debe de dejar tomar una opción en la expansión de cada no terminal.

- **Gramáticas LL(k)**
  - Son un subconjunto de las gramáticas libres de contexto. Permiten un análisis descendente determinista (o sin retroceso), por medio del reconocimiento de la cadena de entrada de izquierda a derecha ("*Left to right*") y que va tomando las derivaciones más hacia la izquierda ("*Leftmost*") con sólo mirar los *k* tokens situados a continuación de donde se halla.
  - Por definición de gramática LL(k):
    - Toda gramática LL(k) es no ambigua.
    - no es recursiva a izquierdas.

3.4.1.3. Implementación de Analizadores LL

- **Análisis descendente sin retroceso.**
  - En el análisis descendente con retroceso se generan formas sentenciales a partir del axioma dando marcha atrás en cuanto se detecta que la forma generada no es viable, (es decir, no conduce a ninguna sentencia del lenguaje). Este proceso de vuelta atrás es lento. Para mejorar la eficiencia del mismo, sería muy útil saber *a priori* qué alternativa del símbolo no terminal es más conveniente usar.
- Ejemplo:
  - gramática

    (1) S $\rightarrow$ cAd
    (2) A $\rightarrow$ bcB
    (3) A $\rightarrow$ a
    (4) B $\rightarrow$ b
  - Análisis de la sentencia ***cad***.
    - Partiendo del axioma, sólo se puede aplicar la regla 1, obteniendo la forma sentencial cAd. Si se compara con la sentencia cad, se observa que ambas comienzan con el caracter c. Por tanto, la subcadena Ad ha de generar el resto de la sentencia, o sea, ad. En este instante existen dos alternativas que se pueden emplear para modificar la forma sentencial, que corresponden a la aplicación de las reglas 2 y 3.
    - La aplicación de la regla 2 provoca la aparición del carácter b al principio de la subcadena restante, mientras que la regla 3 provoca la aparición del carácter a. Por tanto, como la subcadena que falta por generar para producir la sentencia final es ad (empieza por a), puede deducirse que en este instante la regla que debe emplearse es la regla 3, y no la 2.

3.4.1.3. Implementación de Analizadores LL

- **Análisis descendente sin retroceso.**
- El método de análisis que hemos seguido consiste en leer la cadena de entrada de izquierda a derecha, (**L**: *Left to rigth*) utilizando reglas de producción izquierda (**L**: *Left most*) e inspeccionando un (**1**) solo símbolo de la entrada para elegir la regla conveniente. Este análisis se denomina **LL(1)**.
- Hay casos en los que este procedimiento no sirve.
  - Supóngase, por ejemplo, que la gramatica fuese:

    (1) S→ cAd
    (2) A → aB
    (3) A → a
    (4) B → b
  - Al analizar la tira de entrada ***cad***, tras realizar la primera producción obtendríamos la forma sentencial cAd, quedando como subcadena a analizar ad (que comienza con a). Pero ahora hay dos reglas aplicables que comienzan por a (las reglas número 2 y 3). Por tanto, no es posible decidir de forma automática qué regla debe emplearse.
- Si se pretende que el análisis sea sin retroceso, es indispensable que la gramática no tenga ciclos por la izquierda (no sea recursiva por la izquierda).
- No todas las gramáticas admiten un análisis descendente sin retroceso en el que se pueda predecir la alternativa que debe usarse. En el siguiente apartado se verá una condición necesaria y suficiente para que una gramática admita un análisis **LL(1)**.

3.4.1.3. Implementación de Analizadores LL

- **3.4.1.3.1. Transformación de una gramática para su análisis descendente. Construcción de los conjuntos PRIMERO Y SIGUIENTE**
- Funciones asociadas a una gramática independiente del contexto, G que facilitan la construcción de un analizador sintáctico predictivo.
- Defnición: Si α es una cadena de símbolos gramaticales, se llama **PRIMERO(α)** al conjunto de símbolos terminales por los que comienzan las cadenas derivadas de α.
  - Si $\alpha \Rightarrow^* \lambda$, entonces λ también pertenece a PRIMERO(α).
- Defnición: Se define **SIGUIENTE(A)**, para el no terminal A como el conjunto de símbolos terminales que pueden aparecer inmediatamente a la derecha de A en alguna forma sentencial.
  - Esto es, el conjunto de terminales a $\in \Sigma_T$ tales que haya una derivación de la forma $S \Rightarrow^* \alpha A a \beta$ para algunas cadenas α y β.
  - En algún momento de la derivación puede haber existido símbolos entre A y a pero derivaron en λ (se anularon) y desaparecieron.
  - Si A puede ser el símbolo situado más a la derecha en una forma sentencial, entonces $\$ \in$ SIGUIENTE(A) siendo $ el símbolo que delimita la entrada por la derecha.

3.4.1.3. Implementación de Analizadores LL

- **Construcción de los conjuntos PRIMERO Y SIGUIENTE**
- **Cálculo de los Primeros**: Para calcular PRIMERO(X) para todos los símbolos gramaticales X, se aplican las reglas siguientes hasta que no se puedan añadir más terminales a ningún conjunto PRIMERO.
  1. Si X es un terminal, entonces PRIMERO(X) = {X}
  2. Si X $\rightarrow\lambda$ entonces PRIMERO(X) = PRIMERO(X) $\cup$ {λ}
  3. Si X es un no terminal y $\mathbf{X \rightarrow Y_1 Y_2 \ldots\ldots Y_k}$
     - Si existe un i tal que a $\in$ PRIMERO(Y_i) y además

       $Y_1 Y_2 \ldots\ldots Y_{i-1} \Rightarrow^* \lambda$ esto es, si λ está en PRIMERO(Y_1)….. PRIMERO(Y_{i-1})

       entonces:

       PRIMERO(X) = PRIMERO(X) $\cup$ {a}
     - Si $\lambda\in$ PRIMERO(Y_j) para j = 1; 2;…..; k entonces:

       PRIMERO(X) = PRIMERO(X) $\cup$ {λ}

- Por ejemplo, todo los símbolos de PRIMERO(Y_1) sin duda pertenecerán a PRIMERO(X). Si Y_1 no deriva a λ, entonces no se añade nada más a PRIMERO(X), pero si $Y_1 \Rightarrow^* \lambda$, entonces se añaden los elementos de PRIMERO(Y_2) y así sucesivamente.

3.4.1.3. Implementación de Analizadores LL

- Se puede calcular el conjunto PRIMERO para cualquier cadena de la forma $X_1X_2……..X_n$ del siguiente modo:
  - Añadir a PRIMERO($X_1X_2……X_n$) los símbolos $\neq \lambda \in$ a PRIMERO(X_1).
  - Si $\lambda \in$ PRIMERO(X_1), añadir los símbolos$\neq\lambda$ de PRIMERO(X_2).
  - Si λ está tanto en PRIMERO(X_1) como en PRIMERO(X_2), añadir también los símbolos distintos de λ de PRIMERO(X_3) y así sucesivamente.
  - Por último, añadir λ a PRIMERO($X_1X_2……X_n$) si $\forall$i, PRIMERO(X_i) contiene a λ.

- **Ejemplo:**

gramática:

A $\rightarrow$ BCc | gDB

B $\rightarrow$ bCDE | λ

C $\rightarrow$ ca | DaB

D $\rightarrow$ dD | λ

E $\rightarrow$ gAf | c

Primero	
A	g b c d a
B	b λ
C	c d a
D	d λ
E	g c

3.4.1.3. Implementación de Analizadores LL

- **Cálculo de los Siguientes**: para calcular SIGUIENTE(A) $\forall$ A $\in \Sigma_N$ de una gramática, aplicar las siguientes reglas hasta que no se pueda añadir ningún elemento nuevo a ningún conjunto SIGUIENTE.
  1. Incluir $ en SIGUIENTE(S), donde S es el símbolo inicial de la gramática y $ es el delimitador de la entrada por la derecha (el último símbolo de la entrada).
  2. Si hay una producción A $\rightarrow \alpha B \beta$ (con $\beta \neq \lambda$) entonces todo los símbolos de PRIMERO(β) excepto λ se incluyen en SIGUIENTE(B).
  3. Si hay una producción de la forma A $\rightarrow \alpha B$ o bien A $\rightarrow \alpha B \beta$ con $\lambda \in$ PRIMERO(β), es decir, $\beta \Rightarrow^* \lambda$ entonces todos los símbolos de SIGUIENTE(A) se incluyen en SIGUIENTE(B).
  - En la segunda regla se trata de identificar (para calcular sus siguientes) símbolos no terminales que no aparezcan al final de las reglas de producción, mientras que en la tercera regla se han de localizar símbolos no terminales que estén al final de la regla de producción, o bien estén seguidos por cadenas β que derivan en λ.

3.4.1.3. Implementación de Analizadores LL

- Construcción de los conjuntos **PRIMERO** Y **SIGUIENTE**

- Ejemplo:

$A \rightarrow BCc \mid gDB$

$B \rightarrow bCDE \mid \lambda$

$C \rightarrow ca \mid DaB$

$D \rightarrow dD \mid \lambda$

$E \rightarrow gAf \mid c$

	Primero	Siguiente
A	g b c d a	$ f
B	b λ	c d a f $ g
C	c d a	c d g
D	d λ	b g c a $ f
E	g c	c d a f $ g

3.4.1.3. Implementación de Analizadores LL

- **Condiciones LL(1)**
  - Símbolos directores

    Ayudan a decidir qué regla utilizar en cada paso
  - Construcción
    - Conjunto Primero PRIM (α)
    - Conjunto Siguiente SIG (A)
    - Regla
    - DIR (A := α) =
      - **SI $\lambda \in$ PRIM(α) ENTONCES = PRIM (α)– {λ}$\cup$SIG(A)**
      - **SINO = PRIM (α)**

3.4.1.3. Implementación de Analizadores LL

- Condiciones necesarias para ser LL(1)
  - No ambigua
  - Factorizada por la izquierda
  - No recursiva a izquierdas
- **Condición necesaria y suficiente LL(1):** Se dice que una gramática cumple la *condición LL(1)* si para cada par de reglas de la gramática que tengan el mismo antecedente la intersección de sus símbolos directores es **vacía**. Es decir,

  Si se tienen la producciones del no terminal A:

  $A \rightarrow \alpha_1 \mid \alpha_2 \mid ..... \mid \alpha_n$ se debe cumplir que

  1. $\mathbf{PRIM(\alpha_i) \cap PRIM(\alpha_j) = \varnothing} \quad \forall\, i \neq j \quad \alpha_i \in \Sigma^*$
  2. Si $\lambda \in PIM(\alpha_j)$ entonces

     $\mathbf{PRIM(\alpha_i) \cap SIG(A) = \varnothing} \quad \forall\, i \neq j$

3.4.1.3. Implementación de Analizadores LL .

3.4.1.3.2. Anlizador descendente Dirigido por la Sintaxis.

- Dada la **gramática descrita en BNF**, se traduce esas reglas sintácticas a programas:

  Simbolo SLA; /* **Simbolo leido = preanálisis*/**

1. **Programa_Principal;**

```
{ SLA = leer_símbolo();
  S();                  //S=axioma
  Si SLA!=$ entonces
      Error();
}
```

NOTA: $=EOF

2. Los **símbolos terminales** son tokens enviados por el analizador léxico que se traduce en una llamada a la acción Reconocer(t) que comprobará si el símbolo leído coincide o no con el que se le pasa por parámetro. Si es así lee el siguiente, en caso contrario da un mensaje de error.

```
Procedimiento Reconocer (Simbolo terminal)
    BEGIN
        Si (SLA == terminal)
            leer_símbolo();
        sino
            error_sintactico(t);
    END
```

3. Los **símbolos no terminales** son procedimientos, funciones o métodos.

```
Procedimiento Noterminal( );
// Implementación según el siguiente apartado
```

3.4.1.3.1. Anlizador descendente Dirigido por la Sintaxis.

4. Las reglas de producción se traducen en estructuras de control.
   1. La regla sintáctica **A** $\rightarrow$ $\mathbf{B_1B_2.....B_n}$ corresponde a la sentencia compuesta:

```
Procedimiento A()
    BEGIN
        Tratamiento B1;
        Tratamiento B2;
         .......
        Tratamiento Bn
    END
```

   2. La regla de producción **A** $\rightarrow$ $\mathbf{B_1|B_2\ |.....|\ B_n}$ se traduce a:

```
 Procedimiento A()
    CASE token OF
        l1:  Tratamiento B1;
        l2:  Tratamiento B2;
        ..........
        ln:   Tratamiento Bn;
        else: Error()
    END;
```

Donde las etiquetas li son :

- los primeros símbolos terminales para B_i ;
- o en caso que fuese λ el conjunto Siguiente(A)

3.4.1.3.1. Anlizador descendente Dirigido por la Sintaxis.

- Ejemplo:
  - Gramática

    S → xSy| λ
  - Cojuntos PRIM y SIG

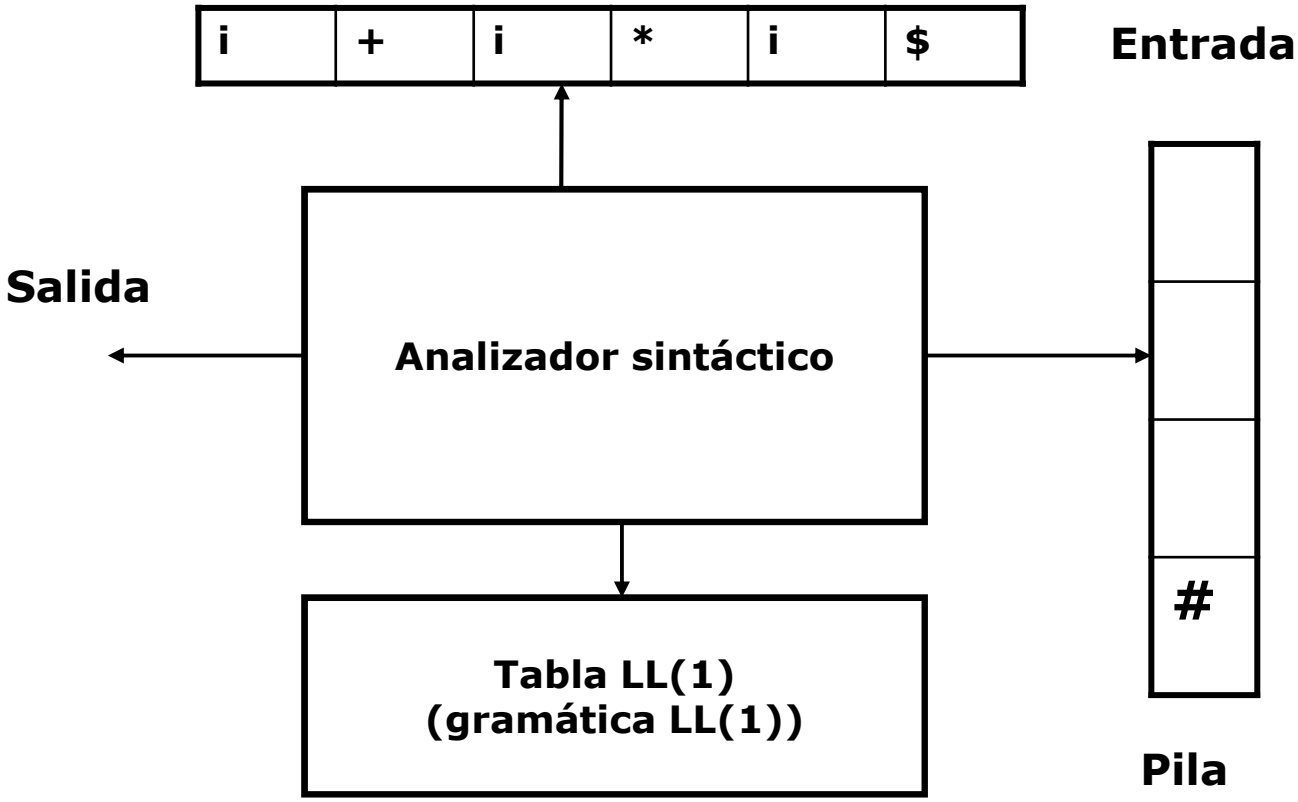
	PRIM	SIG
s	**x,λ**	**y, $**

- Programa:

```
Programa_Principal;
    {   SLA = leer_símbolo();
        S();          //S=axioma
        Si SLA!=$ entonces
          Error();
    }
Procedimiento S()
    CASE SLA OF
        ‘x’:  Reconocer(´x´);
                S();
                Reconocer(´y´);
          ´y´, ´$´ :  /* nada */;
        else: Error()
    END;
```

3.4.1.3.3. Analizador descendente predictivo dirigido por tabla

- Utiliza una tabla (tabla LL(1)) para decidir



Construcción de tablas LL(1)

- Las funciones Primero y Siguiente permiten rellenar siempre que sea posible, las entradas de una tabla de análisis sintáctico predictivo para la gramática.
- Para realizar el análisis sintáctico de una cadena generada por una gramática LL(1) se define un procedimiento basado en el empleo de una pila y una tabla de doble entrada. Este procedimiento consiste en asignar a un par (símbolo leído *(terminal)*, símbolo de pila *(no terminal)*) una regla de la gramática.
- La tabla de análisis se obtiene mediante el siguiente algoritmo:
- Se define la tabla: $TABLA : \Sigma_N \times [\Sigma_T \cup \{\$\}] \rightarrow [\Sigma_N \cup \Sigma_T \cup \{\lambda\}]^*$
- **Algoritmo:**

```
∀ A → α
    ∀ "a" terminal ≠λ ∈ PRIM(α)
        TABLA[A,a]= α
    fin ∀
    Si λ ∈ PRIM(α)
        ∀ "b" terminal ∈ SIG(A)
            TABLA[A,a]= λ
        fin ∀
    Fsi
fin ∀
```

3.4.1.3.2. Analizador descendente predictivo dirigido por tabla.

- Algoritmo de análisis sintáctico LL:

```
Procedimiento Análisis_LL();
  Apilar("#");
  Apilar(S);                    // S = axioma
  Leer(símbolo);                // Preanálisis = símbolo
  Mientras pila_no_vacía hacer
    Caso cima_pila sea
       "terminal":
          Si cima_pila==símbolo entonces
             Desapilar(símbolo);
             Leer(símbolo);
          sino
             Error_sintáctico ()
          fsi
       "No_terminal":
          Si TABLA[cima_pila,símbolo] != "error" entonces
             Desapilar(cima_pila);
             Apilar(TABLA[cima_pila,símbolo]);
          sino
             Error_sintáctico ()
          fsi
    fcaso
  fmientras
  if cima_pila==# entonces
     Desapilar("#");
     Escribir("Cadena Aceptada")
  sino
     Error_sintáctico();
  fsi
fprocedimiento
```

3.4.1.3.2. Analizador descendente predictivo dirigido por tabla. **Ejemplo.**

1. Gramática

   E := E + E
   E := E - E
   E := E * E
   E := E / E
   E := **n**
   E := (E)

3. Primeros y Siguientes

	PRIM	SIG
E	(n	) $
E´	+ - λ	) $
T	n (	+ -) $
T´	* / λ	+ -) $
F	n (	+ - * /) $

2. Gramática equivalente

1. E := TE'
2. E' := +TE'
3. | -TE'
4. | λ
5. T := FT'
6. T' := *FT'
7. | /FT'
8. | λ
9. F := n
10. | (E)

4. Tabla LL(1)

	+	-	*	/	(	)	n	$
E					**1**		**1**	
E´	**2**	**3**				**4**		**4**
T					**5**		**5**	
T´	**8**	**8**	**6**	**7**		**8**		**8**
F					**10**		**9**	

3.4.1.3.2. Analizador descendente predictivo dirigido por tabla. **Ejemplo.**

- Análisis de una cadena
  - Construir una tabla con 3 columnas
    - Pila
    - Entrada procesada
    - Salida/acción
- Ejemplo anterior: análisis para la cadena **n+n*n $**

	PRIM	SIG
E	(n	) $
E´	+ - λ	) $
T	n (	+ -) $
T´	* / λ	+ -) $
F	n (	+ - * /) $

	+	-	*	/	(	)	n	$
E					**1**		**1**	
E´	**2**	**3**				**4**		**4**
T					**5**		**5**	
T´	**8**	**8**	**6**	**7**		**8**		**8**
F					**10**		**9**	

1. E := TE'
2. E' := +TE'
3. | -TE'
4. | λ
5. T := FT'
6. T' := *FT'
7. | /FT'
8. | λ
9. F := n
10. | (E)

pila	entrada	acción
λ	n+n*n$	Apilar("#");
#	n+n*n$	Apilar(E);
E#	n+n*n$	*E → T E'* **/* Desapilar(cima_pila);** **Apila(**TABLA[cima_pila,símbolo]**);*/**
TE'#	n+n*n$	*T → FT'*
FT'E'#	n+n*n$	*F* → n
nT'E'#	n+n*n$	Reconocer(n) **/* Desapilar(símbolo;** Leer(símbolo);*/
T'E'#	+n*n$	*T'* → λ
E'#	+n*n$	*E' → +TE'*
T'E' E'#	+n*n$	*T'* → λ
E' E'#	+n*n$	*E' → +TE'*
+TE'E'#	+n*n$	Reconocer(+)
TE'E'#	n*n$	*T → FT'*
FT'E'E'#	n*n$	*F* → n
nT'E'E'#	n*n$	Reconocer(n)
T'E'E'#	*n$	*T' → *FT'*
*FT'E'E'#	*n$	Reconocer(*)
FT'E'E'#	n$	*F* → n
nT'E'E'#	n$	Reconocer(n)
T'E'E'#	$	*T'* → λ
E'E'#	$	*E'* → λ
E'#	$	*E'* → λ
#	$	Desapilar("#");
λ	λ	**Aceptar**

3.4.1.3.4. Tratamiento de los Errores en Análisis Descendente

- Tratamiento de errores:
  - Detectar sólo el primero
  - Detectar el máximo (razonable)
    - Sincronizar después del error
    - Prever errores típicos y cambiar la gramática a otra que los acepte dando warnings.
    - Corregir el error: Decidir cual ha sido el error y reconstruir el programa fuente eliminando dicho error
- Consideraciones:
  - No se pueden solucionar correctamente todos los casos.
  - Como mínimo tratar correctamente los errores más comunes.
  - Minimizar:
    - Error no detectados
    - Falsas alarmas