Lección 4: Análisis Sintáctico LL(1)

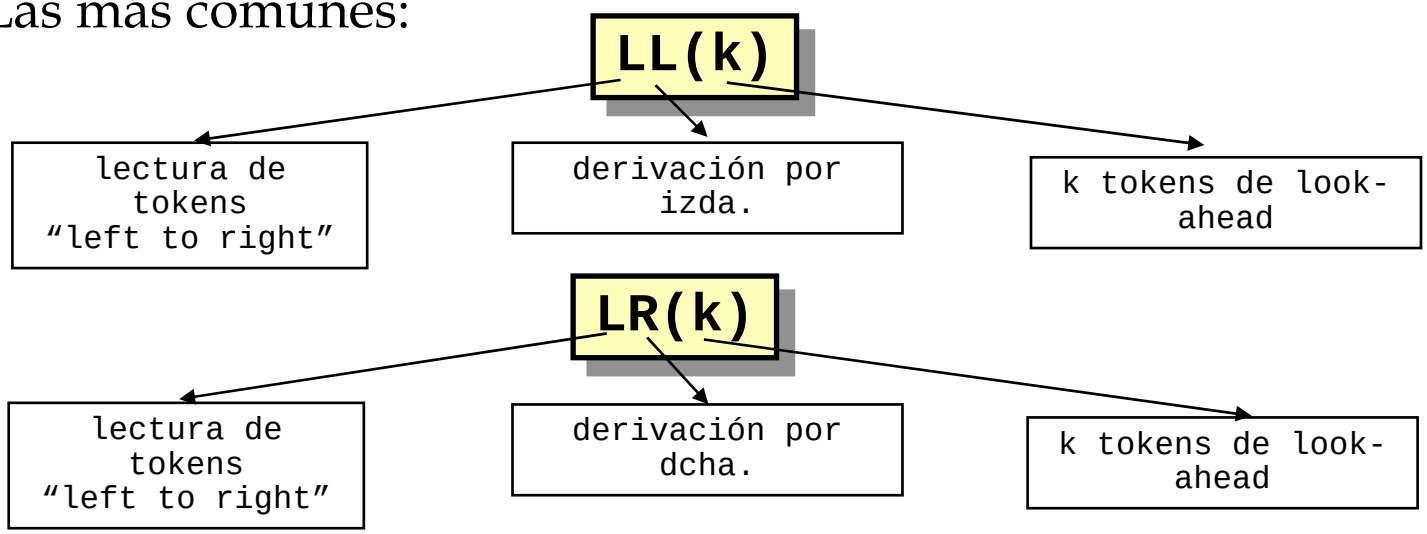
1) Estrategias para el Análisis Sintáctico
2) Análisis Sintáctico descendente
3) Factorización a izda. de gramáticas
4) Eliminación de la recursividad a izda.
5) Construcción de Analizadores Sintácticos predictivos
6) Construcción de Analizadores Sintácticos predictivos no recursivos
7) Construcción de una tabla para el análisis sintáctico predictivo
8) Sobre recuperación de errores en el análisis sintáctico predictivo

Estrategias para analizadores sintácticos

- El analizador sintáctico debe verificar si la entrada corresponde a alguna derivación de S
- Primera aproximación: "a lo bestia"
  - generar todas las derivaciones de S posibles
  - comprobar si la entrada corresponde con alguna de ellas
- Inviable, incluso para gramáticas muy sencillas
- Son necesarias estrategias más "astutas"
- Las estrategias son de dos categorías:
  - análisis DESCENDENTE (Top-Down)
    - » construye el árbol desde la raíz (S) hasta llegar a las hojas
  - análisis ASCENDENTE (Bottom-Up)
    - » construye el árbol desde las hojas hacia la raíz (S)

Estrategias para analizadores sintácticos

- A distintos tipos de analizadores se les da nombres en función a varios elementos:
  - lectura de tokens (izda. a dcha., viceversa)
  - método de derivación (izda. o dcha.)
  - número de tokens de pre-análisis (look-ahead)
- Las más comunes:



Estrategias para analizadores sintácticos

- Propiedades deseables en un analizador sintáctico
  - eficiente
    - » en general, se buscarán polinomiales en el tamaño del programa a reconocer
  - determinar la acción reconociendo unos pocos tokens de entrada
    - » como máximo, un ***k*** preestablecido
    - » en la práctica, suele ser ***k***=1
  - no necesitar "marcha atrás" (backtracking)
    - » i.e., evitar decisiones
    - » las acciones semánticas son difíciles de deshacer
  - no ocupar mucha memoria
    - » actualmente, esto no es tan importante

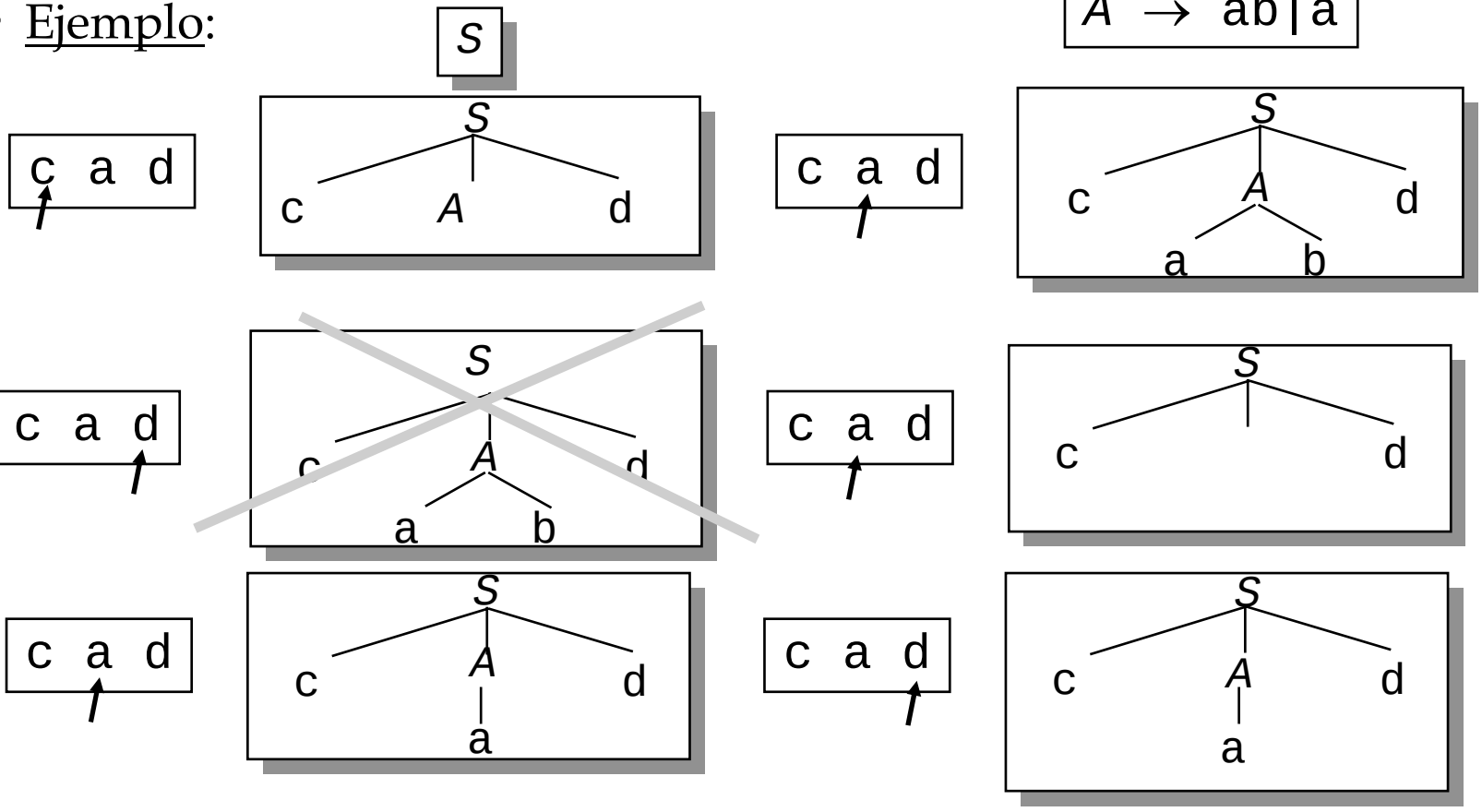
- Básicamente, es un intento de encontrar una ***derivación por la izda.***
- Desde el punto de vista del árbol de sintaxis, correspondería a un ***recorrido en pre-orden***
- Método:
  - Objetivo: reconocer S
  - Método:
    - » se divide el objetivo en subobjetivos (de acuerdo a las producciones)
    - » se trata de cumplir cada subobjetivo
    - » se sigue con cada subobjetivo, hasta que se encuentren concordancias de terminales (o se detecte un error)

cad

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

- Ejemplo:



- Sobre implementación de analizadores descendentes
  - la forma más natural es asociar un procedimiento con cada no terminal
  - cada procedimiento:
    - » comprueba la corrección del token en estudio con el que el propio procedimiento representa
      - incorrecto: mensaje de error y/o estrategia de recuperación
    - » hacer avanzar al siguiente token
  - afinando un poco más:
    - » no terminal: genera invocaciones
    - » terminal: concordancia entre terminales
      - error sintáctico ó
      - avanzar a token siguiente

$S \rightarrow cAd$

$A \rightarrow a$

- Un ejemplo muy sencillo

```
int elToken;
int yylex(){
   return(getchar());
}

void terminal(int t){
   if(elToken==t)
      elToken=yylex();
   else
      errorSintactico();
}
void A(){
   terminal((int)'a');
}
```

```
void S(){
   terminal((int)'c');
   A();
   terminal((int)'d');
}
void main(){
   elToken=yylex();
   S();
}
```

¿Qué pasa con

$S \rightarrow cAd$

$A \rightarrow a|ab$

?

$S \rightarrow cAd$

$A \rightarrow a|ab$

- Una solución predictiva

```
int elToken;
int yylex(){
   return(getchar());
}

void terminal(int t){
   if(elToken==t)
      elToken=yylex();
   else
      errorSintactico();
}
void A(){
   terminal((int)'a');
   if(elToken==(int)'b')
       terminal((int)'b');
}
```

```
void S(){
   terminal((int)'c');
   A();
   terminal((int)'d');
}
void main(){
   elToken=yylex();
   S();
}
```

preanálisis

- Ejemplo: Se desea construir un analizador sintáctico para la siguiente gramática

instruccion → **identificador opas** *expr* **';'**

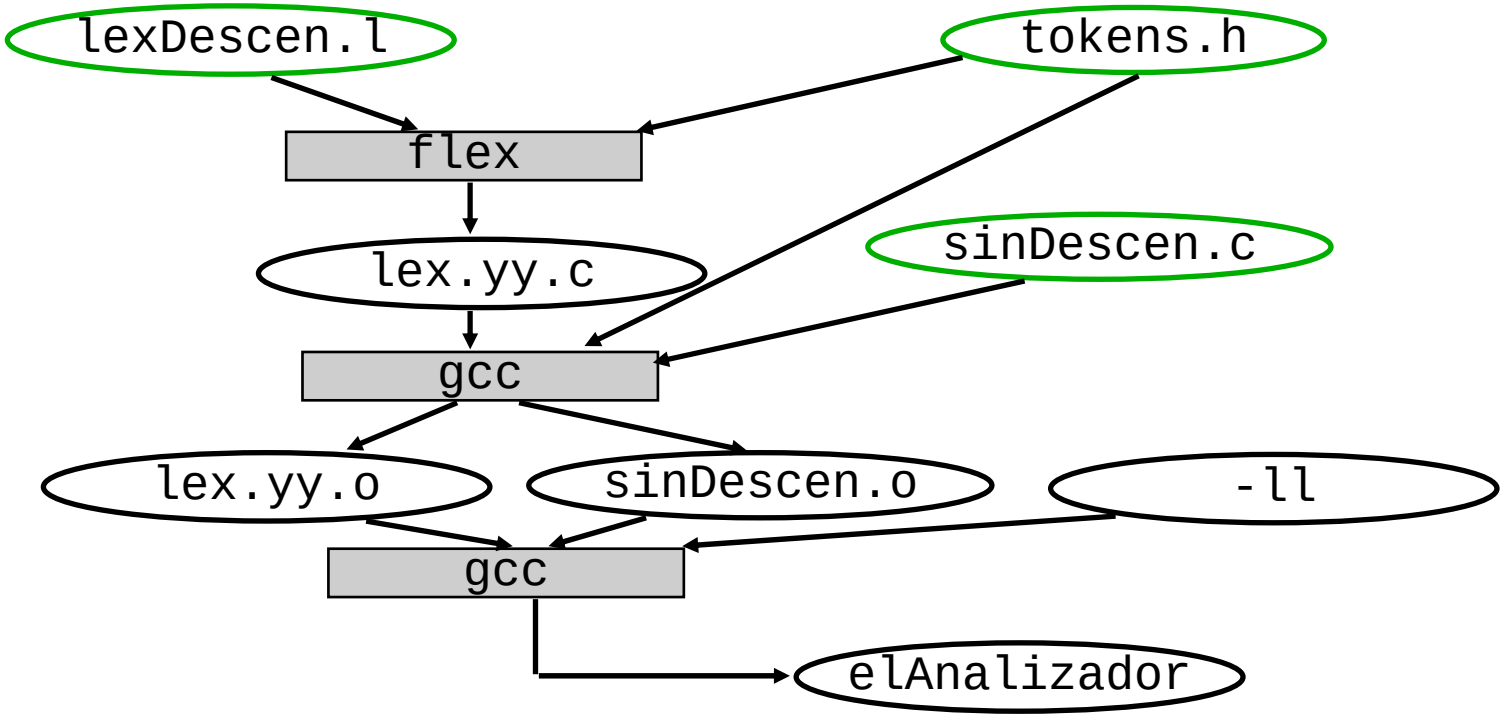
expr → *termino expr'*

expr' → **'+'** *termino expr'* | ε

termino → **identificador** | **'('** *expr* **')'**

- donde:
  - "`opas`" es un terminal de la gramática que representa al operador de asignación.
    - » Corresponde al lexema ":="
  - "`identificador`"
    (letra seguido de letras o dígitos)

- Como paso previo, necesitamos construir el analizador léxico
- Esquema del proceso:



Análisis descendente. Un ejemplo

tokens.h

```
enum{ASIGNACION=257,IDENTIFICADOR};
```

lexDescen.l

```
%{
#include <stdio.h>
#include "tokens.h"
extern char yytext[]; /*ó extern char *yytext*/
%}
        /*EXPRESIONES REGULARES*/
separador     [\t \n]+ /*tabs,blancos*/
letra         [a-zA-Z]
digito        [0-9]
identificador {letra}({letra}|{digito})*
%%
{separador}       {/*me los como*/}
":="              {return(ASIGNACION);}
{identificador} {return(IDENTIFICADOR);}
.                 {return(yytext[0]);}
%%
/* no hay funciones de usuario */
```

Análisis descendente. Un ejemplo

- Para el analizador sintáctico, construi remos un

`analizador sintáctico descendente recursivo`

- Ideas básicas del método:
  - construir un conjunto de procedimientos (recursivos si necesario)
    - » uno por cada no terminal de la gramática
  - cada procedimiento:
    - » determina la producción a aplicar
    - » invoca los proc. correspondientes
    - » detecta error o pasa al siguiente token
  - la secuencia de invocaciones a los procedimientos define implícitamente el árbol de sintaxis
- En este ejemplo funciona bien por las razones que veremos, pero no siempre es tan sencillo

Análisis descendente. Un ejemplo

sinDescen.c

```
#include <stdio.h>
#include "tokens.h"
int elToken;            /*global con sig. token*/
void terminal(int token{
  if(elToken==token)
    elToken=yylex();
  else
    hayError(...INFO(elToken) inesperado...);
}
void instruccion(){
  terminal(IDENTIFICADOR);
  terminal(ASIGNACION);
  expresion();
  terminal((int) ';');
}
void main(){
  elToken=yylex();
  instruccion();
}
```

instruccion →
identificador
opas
expresion ';'

Análisis descendente. Un ejemplo

sinDescen.c

```
void termino(){
  if(elToken==IDENTIFICADOR)
    terminal(IDENTIFICADOR);
  else if(elToken=='('){
    terminal((int) '(');
    expresion();
    terminal((int) ')');
  }else
    hayError("Se esperaba un IDENTIFICADOR o '('");
}
```

termino → **identificador**
| **(** expresion **)**

Análisis descendente. Un ejemplo

expresion → *termino expresionP*

sinDescen.c

```
void expresion(){
    termino();
    expresionP();
}


void expresionP(){
  if(elToken=='+'){
    terminal((int) '+');
    termino();expresionP();
  } /*else: corresponde a ε */
}
```

expresionP → + *termino expresionP* | ε

- En general, el método puede presentar incovenientes:
  - necesidad de backtracking: cuando la regla elegida no es la correcta "devolver" a la entrada todos los tokens recorridos
    - » Ejemplo:

```
instIF →
    tkIF expr tkTHEN insts
  | tkIF expr tkTHEN insts tkELSE insts
```

  - recursividad a izda.: si alguna regla se define con recursividad a izda., el analizador sintáctico construído entra en un bucle infinito
    - » Ejemplo:

```
expr → expr + term | termino
```

- Soluciones:
  - factorización a izda.
  - eliminación de recursividad a izda.

- Consideremos la gramática $A \rightarrow \alpha C \mid \alpha B$

- Si encontramos una cadena derivada de α, ¿Qué producción aplicar?
  - ¿ Si sigo αC y era αB ?
  - ¿ Si sigo αB y era αC ?
- Idea: retrasar la decisión, de manera que siempre se tome la buena decisión

$$A \rightarrow \alpha A'$$
$$A' \rightarrow C \mid B$$

- Ambas gramáticas generan el mismo lenguaje

- En analizadores descendentes, la recursividad a izda. da lugar a bucles infinitos
- Ejemplo:

```
instruccion → identificador opas expr ';'
expr → expr '+' termino
     |  termino
termino → identificador
        | '(' expr ')'
```

- Cuando se trata de derivar

```
velocidad:=(vel1+vel2)+vel3;
```

la invocación "`expresion()`" se hace recursivamente sin consumir nuevos tokens, dando un bucle infinito

- Por suerte, la rec. directa a izda. se puede eliminar mediante una transformación de la gramática

$$A \rightarrow A\alpha \mid \beta$$

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

- Ambas gramáticas son equivalentes

- Aplicando la transformación a la gramática anterior, obtenemos

```
instruccion → identificador opas expr ';'
expr → termino expr'
expr' → '+' termino expr' | ε
termino → identificador | '(' expr ')'
```

- Sin embargo, sólo eliminamos la rec. izda. directa
- ¿Qué pasa con

$$A \rightarrow B\alpha \mid \alpha$$
$$B \rightarrow AB \mid \beta$$

?

- El siguiente algoritmo resuelve el problema [AhSU 90]

```
Algoritmo  eliminaRecIzda(E G:GLC; S G':GLC)
--Pre: G=(N,T,P,S) sin "ciclos" (A ⇒+ A) ni
--   producciones ε (A → ε) ∧ N={A1,...,An}
--Post: G' ≅ G, sin rec. a izda.

Principio
  G':=G
  Para i:=1 hasta n
    Para j:=1 hasta i-1
       --Aj → δ1|...|δk son las
       --    producciones actuales de Aj
      sustituir en P'
            Ai → Ajα por Ai → δ1α|...|δkα
    FPara
    eliminar rec. inmediata de Ai
  FPara
Fin
```

Para tratamiento de ciclos y producciones ε, ver [HoUl 79]

Sep 97/98

Ejercicio 4 (1.5 ptos.): Considerar el siguiente código C que corresponde a un analizador sintáctico descendente recursivo. Determinar, a partir del propio código, cuál es la gramática, y decir si el analizador funcionará correctamente.

```
enum token …
extern enum token getToken(void);
enum token tok;
void advance(){
   tok=getToken();
}
```

```
void eat(enum token t){
   if(tok==t)
     advance();
   else
     error();
}
```

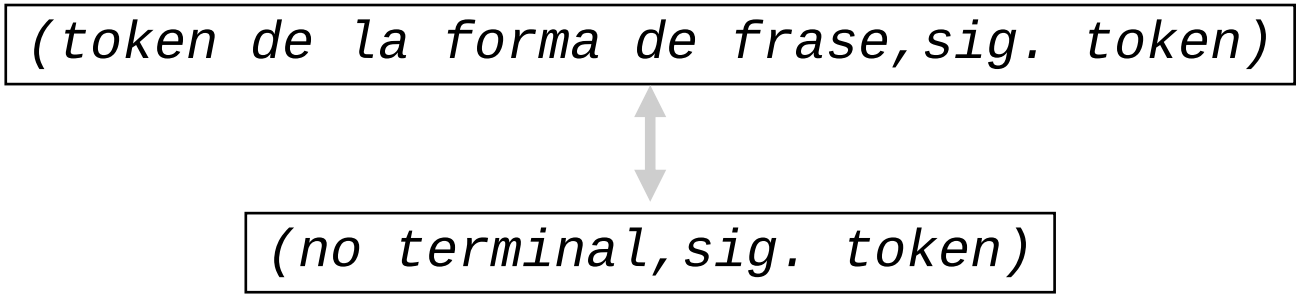
Análisis descendente. Recursividad a izda.

```
void S(){
   switch(tok){
     case IF: eat(IF);E();eat(THEN);
              S(); eat(ELSE);S();break;
     case BEGIN: eat(BEGIN);S();L();break;
     case PRINT: eat(PRINT);E();break;
     default:  error();
   }
}
void L(){
   switch(tok){
      case END: eat(END); break;
      case SEMI: eat(SEMI);S();L();break;
      default: error();
   }
}
void E(){
   eat(NUM); eat(EQ); eat(NUM);
}
```

Analizadores sintácticos predictivos

- Un analizador sintáctico predictivo es un analizador sintáctico descendente recursivo que NO necesita marcha atrás
- Condición exigible a una gramática para que sea posible:
  - que en todo momento, a partir del siguiente token de entrada, sea posible determinar qué regla aplicar
- Esto es posible cuando (una de dos):
  - ningún no terminal tiene alternativas en su parte derecha
    - » gramáticas poco interesantes
  - no terminales con alternativas que empiecen por distintos símbolos
- Aunque hay más casos en que también es posible

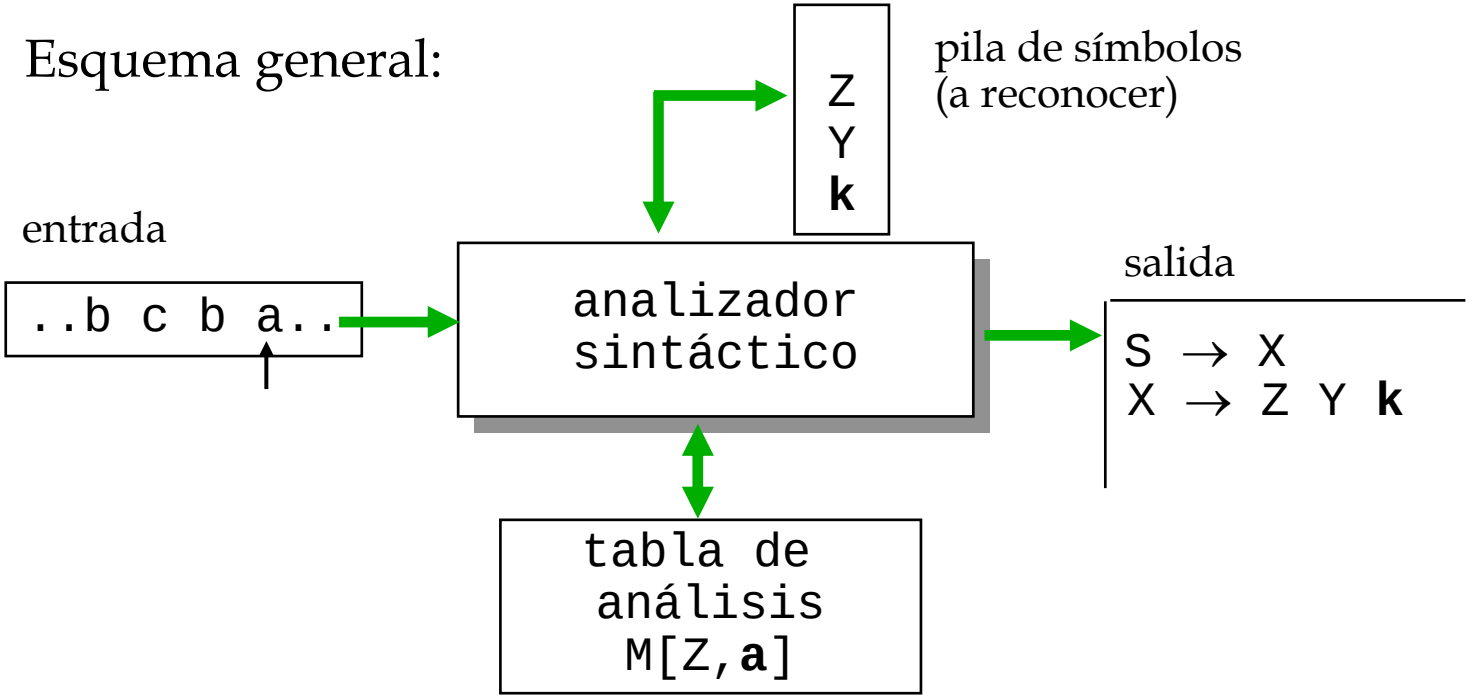
- El hecho de que haya reglas recursivas hace que el analiziador predictivo implementado directamente sea recursivo
- Sin embargo, la recursividad se puede evitar mediante el uso explícito de una pila
- Un analizador predictivo sabe, a partir del símbolo que está analizando y el siguiente símbolo de entrada, qué regla aplicar
- Por lo tanto, se puede hacer corresponder a una tabla de doble entrada:



- Esta forma de implementación da lugar

`anal. sintácticos dirigidos por tabla`

- Esquema general:



- Esquema de funcionamiento:

```
Pre: M:Nx(T∪{$})→2^P:tabla de anál. sint.
     la sec. de tokens termina con $
Post: sec. de prod. aplicadas o error

Variables p:pila de símbolos gramaticales;sigTok,X:token
Principio
   pilaVacia(p);sigTok=yylex();push(p,$,S)
   Repetir
      X=cima(p)
      Si X es terminal ∨ X=$ entonces
         Si X=sigTok entonces
            pop(p); sigTok=yylex()
         Si no
            errorSintáctico()
         FSi
      Si no
         ....
   Hasta Que X=$
Fin
```

Pre: M:Nx(T∪{**$**})→$2^P$:tabla de anál. sint.
la sec. de tokens termina con **$**
Post: sec. de prod. aplicadas o error

```
Variables p:pila de símbolos gramaticales;sigTok,X:token
Principio
   Repetir
      ...
      Si no
         Si M[X,sigTok]= X→Y1... Yk entonces
            pop(p)
            push(p,YkYk-1... Y1)
            emitir producción X→Y1... Yk
         Si no
            errorSintáctico()
         FSi
      FSi
   Hasta Que X=$
Fin
```

- Ejemplo: reconocer **(x;(x))** para

$$
\begin{aligned}
S &\rightarrow (\; A \;) \\
A &\rightarrow CB \\
B &\rightarrow ; \; A \mid \varepsilon \\
C &\rightarrow x \mid S
\end{aligned}
$$

	(	)	;	x	$
S	S→(A)				
A	A→CB			A→CB	
B		B→ ε	B→;A		
C	C→S			C→x	

Analizadores sintácticos predictivos no rec.

pila	entrada	producción usada (salida)
$S	(x;(x))$	S→(A)
$)A(	(x;(x))$	
$)A	x;(x))$	A→CB
$)BC	x;(x))$	C→x
$)Bx	x;(x))$	
$)B	;(x))$	B→;A
$)A;	;(x))$	
$)A	(x))$	A→CB
$)BC	(x))$	C→S
$)BS	(x))$	S→(A)
$)B)A(	(x))$	
$)B)A	x))$	A→CB
$)B)BC	x))$	C→x
$)B)Bx	x))$	
$)B)B	))$	B→ε
$)B)	))$	
$)B	)$	B→ε
$)	)$	
$	$	

Construcción de una tabla para el análisis

- Lo fundamental para la construcción de un analizador sintáctico predictivo no rec. **construcción de la tabla**
- Idea básica: uso de las funciones **"primero" y "siguiente"**
- Objetivo: utilizar dichas funciones para construir la tabla de análisis predictivo
- Es más: una gramática es LL(1) ssi la tabla construída tiene una única salida para cada entrada

Construcción de una tabla para el análisis

- Sea **G=(N,T,S,P)** una gramática
  Sea α una cadena de símbolos de la gramática: $\alpha \in (N \cup T)^*$
- **PRI(α):** terminales que pueden comenzar una forma de frase derivable de α

$$PRI(\alpha)=\{a \in T \mid \alpha \Rightarrow^* a\beta\} \cup$$
$$(\text{Si } \alpha \Rightarrow^* \varepsilon \text{ entonces } \{\varepsilon\} \text{ si no } \varnothing)$$

- Importante: para (empezar a) reconocer α, el siguiente token a procesar deberá estar en PRI(α)

Construcción de una tabla para el análisis

- Paso 1: Calcular los conjuntos PRI(X) para cualquier símbolo gramatical $X \in N \cup T$

- Paso 2: **Sea $X_1 \ldots X_n \in (N \cup T)^*$**
  - **Añadir a PRI($X_1 \ldots X_n$) los símbolos de PRI(X_1)\{ε}**
  - **Si ε ∈ PRI(X_1)**
    añadir a PRI($X_1 \ldots X_n$) los símbolos de PRI(X_2) \{ε}
  - **Si ε ∈ PRI(X_1) ∩ PRI(X_2)**
    añadir a PRI($X_1 \ldots X_n$) los símbolos de PRI(X_3) \{ε}
  - **Si ε ∈ PRI(X_1) ∩ PRI(X_2) ∩ PRI(X_3)**
    añadir a PRI($X_1 \ldots X_n$) los símbolos de PRI(X_4) \{ε}
  - **.....**
  - **Si ε ∈ PRI(X_1)∩ .... ∩ PRI(X_n)**
    añadir ε a PRI($X_1 \ldots X_n$)

Construcción de una tabla para el análisis

- Paso 1: aplicar el siguiente algoritmo

```
--Pre: X ∈ N∪T
--Post: calcula PRI(X)

Repetir
  Si X∈T
      añadir X a PRI(X)
  FSi
  Si X → ε es una producción
      añadir ε a PRI(X)
  FSi
  Si X → Y1...Yk es una producción
     Si a∈PRI(Yj) ∧ ε∈PRI(Y1)∩... ∩PRI(Yj-1)
          añadir a a PRI(X)
     FSi
     Si ε ∈PRI(Y1)∩... ∩PRI(Yk)
          añadir ε a PRI(X)
     FSi
  FSi
Hasta Que no se añada nada a ningún PRI
```

- Ejemplos:

$S \rightarrow A\ B\ c$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b \mid \varepsilon$

PRI(S)={a,b,c}

PRI(A)={a,ε}

PRI(B)={b,ε}

PRI(a)={a}

PRI(b)={b}

PRI(c)={c}

PRI(AB)={a,b,ε}

$S \rightarrow a\ S\ e \mid B$

$B \rightarrow b\ B\ e \mid C$

$C \rightarrow c\ C\ e \mid d$

PRI(S)={a,b,c,d}

PRI(B)={b,c,d}

PRI(C)={c,d}

PRI(a)={a}

PRI(b)={b}

....

Construcción de una tabla para el análisis

- Sea G=(N,T,S,P) una gramática
- Sea A un no terminal ($A \in N$)
- **SIG(A)** será el conjunto de todos aquellos terminales que pueden aparecer detrás de A en alguna forma de frase

$$\begin{aligned} SIG(A)=\{a \in T \mid S \Rightarrow^{+} ...Aa...\} \cup \\ (Si\ S \Rightarrow^{+} \alpha A \\ \quad entonces\ \{\$\} \\ \quad si\ no\ \ \emptyset \\ ) \end{aligned}$$

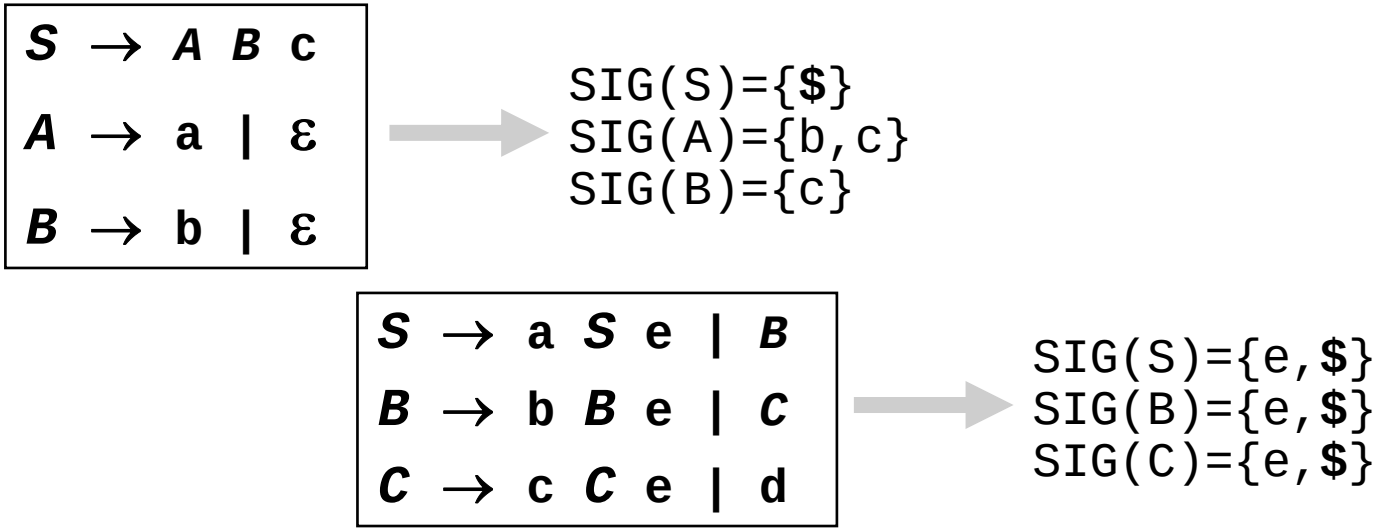
- Importante: **SIG(A)=**conjunto de terminales que pueden indicar (fin del) reconocimiento de una regla que tenga A en la parte izda.

- Construcción de los conjuntos SIG

```
--Pre: TRUE
--Post: calcula SIG(A) para todo A ∈ N

añadir $ a SIG(S)
Repetir
  Si A → αBβ es una producción
          añadir PRI(β)\{ε} a SIG(B)
  FSi
  Si    A → αB es una producción
     ∨ (A → αBβ con ε∈PRI(β) es producción)
          añadir SIG(A) a SIG(B)
  FSi
Hasta Que no se añada nada a ningún SIG
```

- Ejemplos:


S → A B c
A → a | ε
B → b | ε
SIG(S)={$}
SIG(A)={b,c}
SIG(B)={c}
S → a S e | B
B → b B e | C
C → c C e | d
SIG(S)={e,$}
SIG(B)={e,$}
SIG(C)={e,$}

Construcción de una tabla para el análisis

- Construcción de la tabla:
  - filas indexadas por no terminales
  - columnas indexadas por terminales ($\cup$ {**$**})
- Ideas a seguir para la producción

$$A \rightarrow \alpha$$

  - Si $a \in PRI(\alpha)$, cuando entra **a** se expande A por α
  - Si $\alpha = \varepsilon$ ó $\alpha \Rightarrow^* \varepsilon$
    - » expandir A por α cuando llegue algo de `SIG(A)`

Construcción de una tabla para el análisis

Pre: Una gramática **G=(N,T,S,P)**
Post: M: tabla de anál. desc.

```
Principio
  Para cada (n,t)∈Nx(T∪{$})
   M[n,t]:= ∅ /*¿Conjunto?*/
  FPara
  Para cada producción X → α
   Para cada a∈PRI(α)
     M[X,a]:= M[X,a]∪{X → α}
   FPara
  FPara
  Para cada (n,t)∈Nx(T∪{$})
    Si M[n,t]=∅ ent M[n,t]=ERROR FSi
  FPara
Fin
```

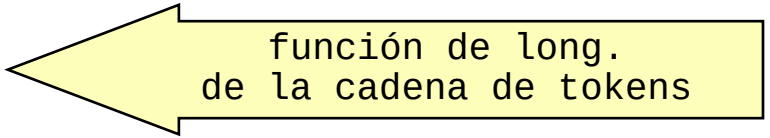
```
Si ℰ∈PRI(α) entonces
  Para cada b∈T∩SIG(X)
    M[X,b]:= M[X,b]∪{X → α}
  FPara
  Si $∈SIG(X) entonces
    M[X, $]:= M[X, $]∪{X → α}
  FSi
FSi
```

Una gramática es LL(1) cuando en la tabla construída según el método anterior cada elemento es una de las dos cosas siguientes:
- ***ERROR**
- ***un conjunto con una única producción**

gramática LL(1)

- Aternativamente, G es LL(1) ssi para toda producción $A \rightarrow \alpha \mid \beta$
  - α y β no pueden derivar cadenas que comiencen por un mismo terminal `a`
  - $\neg((\alpha \Rightarrow^* \varepsilon) \wedge (\beta \Rightarrow^* \varepsilon))$
  - Si $\beta \Rightarrow^* \varepsilon$, α no deriva ninguna cadena que comience con un elemento de $T \cap SIG(A)$

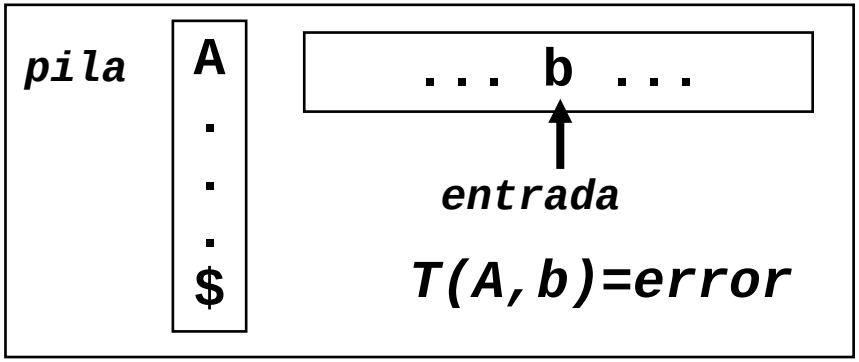
- Notar las ventajas de LL(1):
  - por cada token de entrada, sabemos exactamente **qué** producción se debe aplicar
  - **no** hay backtracking
  - asegura un anal. descendente **correcto**
  - no son ambíguas
  - **complejidad** de los anal. LL(1) :
    - » **lineal** en tiempo
    - » **lineal** en espacio



- ¿Si la gramática no es LL(1)?
  - intentar transformarla en LL(1)
  - no siempre es posible
  - aunque aparecen con mucha frecuencia

- También existen gramáticas LL(k) que emplean una generalización de las técnicas presentadas

- ¿Cuál es el objetivo de la recuperación de errores?
  - no abortar la compilación
  - informar de las decisiones tomadas
- Situación de error:



- ¿Cómo recuperarse de la situación?
  - **recuperación**: eliminar símbolos de la entrada o de la pila hasta llegar a un estado que pueda seguir
  - **reparación**: "reparar" la entrada del usuario

Sobre recuperación/reparación de errores

- Recuperación
  - **modo pánico**: eliminar símbolos de la entrada hasta encontrar alguno específico del **conjunto de sincronización**
- Reparación:
  - supongamos una entrada en el estado $\alpha \mathbf{t} \beta$, donde α es lo ya analizado y $\mathbf{t}$ el siguiente token
  - tres actuaciones posibles:
    - » modificar α
    - » insertar un δ esperando que $\alpha \delta \mathbf{t} \beta$ sea del lenguaje
    - » eliminar $\mathbf{t}$ esperando que sea $\alpha\beta$ del lenguaje

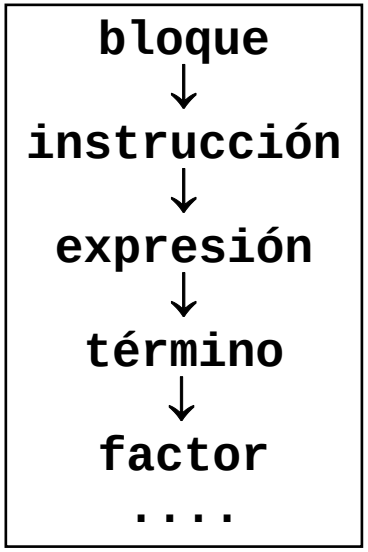
```
...

x := 27
z ;
y := 29 ;
...
```

```
insts  → insts ';' inst
insts  → inst
inst   → instMQ | instAS | ...
instAS → tkID tkOPAS exp
instMQ → tkMQ exp insts
...
```

- al encontrar "z" podemos tratar de arreglarlo como:
  - *Se esperaba ";"*
    - por lo que lo "insertamos". Sin embargo ....
  - *Se esperaba ";"*
    - por lo que seguimos hasta encontrarlo. Sin embargo ....

- No hay un método universal, por lo que se aplican ciertas reglas heurísticas
- Para cada no terminal, definamos **`sinc(A)`**
- Veamos algunos elementos a añadir en **`sinc(A)`**
  - añadir **`SIG(A)`** a **`sinc(A)`**
  - usar la jerarquía del lenguaje
    - » añadir el conjunto de sinc de un nivel a los inferiores



- Ejemplo:

```
insts  → inst insts'
insts' → inst insts' | ε
inst   → instMQ | instAS | ...
instAS → tkID tkOPAS exp ';'
instMQ → tkMQ '(' exp ')' insts
         ...
```

```
x = 3
While (x > y) {
    ....
}
```

- para **A∈N**, añadir **PRI(A)** a **sinc(A)**
  - » Seguir análisis por A cuando llegue uno de ellos
- si **A** $\Rightarrow^*$ **ε**, **cima(p)=A**,
  - » disparar **A** $\Rightarrow^*$ **ε**
- si **cima(p)=x** y **x<>sigToken** (x es terminal)
  - » eliminar **x** de la pila
  - » dar mensaje "***x ha sido insertado***"

Es una inserción "virtual"

Sobre recuperación/reparación de errores

```
procedure pruebaErrores1 is
   x: integer;
   y: integer := 1;
begin
   x := 3
   while x>y loop
      x := x-1;
   end loop;
   x :=  ;
end;
```

gnatmake

```
5:15: missing ";"
9:13: missing operand
```

```
int main(){
   int x,
       y = 1;

   x = 3
   while(x>y)
      x = x-1;
   x = ;
}
```

gcc

```
6: parse error before `while'
8: parse error before `;'
```

Sobre recuperación/reparación de errores

```
procedure pruebaErrores2 is
   x: integer;
   y: integer := 1;
begin
   x := 3  z;
   while x>y loop
      x := x-1;
   end loop;
   x :=  ;
end;
```

gnatmake

```
10:10: binary operator expected
14:08: missing operand
```

```
int main(){
   int x,
       y = 1;

   x = 3 z;
   while(x>y)
      x = x-1;
   x = ;
}
```

gcc

```
5:parse error before `z`
8:parse error before `;`
```

```
int main(){
   int x,
       y = 1;

   x = 3
   while(x>y)
      x = x-1;
   x = ;
}
```

lint

```
line 6: error 1000: Unexpected symbol: "while".
line 7: error 1000: Unexpected symbol: "x".
line 8: error 1000: Unexpected symbol: ";".
line 5: error 1533: Illegal function call.
line 5: warning 714.
line 5: error 1549: Modifiable lvalue required for assignment operator.
line 1: warning 517: Function returns no value.
line 1: warning 845: errors seen skipping the code including function.
```

Sobre recuperación/reparación de errores

```
NAME
  lint - a C program checker/verifier
SYNOPSIS
  lint [options] file ...
DESCRIPTION
  lint attempts to detect features in C program files that are likely to
  be bugs, non-portable, or wasteful.  It also checks type usage more
  strictly than the compilers.  Program anomalies currently detected
  include unreachable statements, loops not entered at the top,
  automatic variables declared but not used, and logical expressions
  whose value is constant.  Usage of functions is checked to find
  functions that return values in some places and not in others,
  functions called with varying numbers or types of arguments, and
  functions whose values are not used or whose values are used but none
  returned.
```

Jun. 96

•**Ejercicio 1:** Considérese la siguiente gramática:

```
S →  [ S
   |  S1
S1→  [a]
```

•1.1) (1 pto.) Establecer el lenguaje que genera. Es necesario que se justifique, lo más formalmente posible, la respuesta

•1.2) .....

•1.3) (1 pto.) ¿Se trata de una gramática LL(1)?

•1.4) (0.5 ptos.) Si no es SLR(1), dar una gramática equivalente que sí lo sea. Análogamente, si no es LL(1) dar una gramática equivalente que sí lo sea..

Feb. 96

Ejercicio 2 (5 ptos.) : Considérese la siguiente gramática (en negrita los terminales):

S → **(** L **)**

S → **a**

L → L **,** S

L → S

2.1 (0.5 ptos.) ¿Qué lenguaje genera?

2.2 .....

2.3 (2 ptos.) Evidentemente, se trata de una gramática no LL(1). Transformarla en una equivalente que sí lo sea, y contruir la tabla del análisis sintáctico LL(1)

Sep. 98

Ejercicio 3 (5.5 ptos. Sep. 98): Considérese la siguiente gramática:

G:

$S \rightarrow \mathbf{u}BD\mathbf{z}$
$B \rightarrow B\mathbf{v}$
$B \rightarrow \mathbf{w}$
$D \rightarrow EF$
$E \rightarrow \mathbf{y}$
$E \rightarrow \varepsilon$
$F \rightarrow \mathbf{x}$
$F \rightarrow \varepsilon$

1) Construir las tablas del análisis sintáctico SLR, y determinar si se trata de una gramática SLR o no
2) Comprobar que se trata de una gramática no LL(1). Transformarla en una gramática LL(1), y calcular su tabla de análisis LL(1)

Feb. 98

Ejercicio 3 (3.0 ptos.): Considerar la siguiente gramática (en negrita los terminales):

S → P M
P → E
  | **q**
M → **n** E
E → **,** O
  | E
O → **u**
  | **u,**O

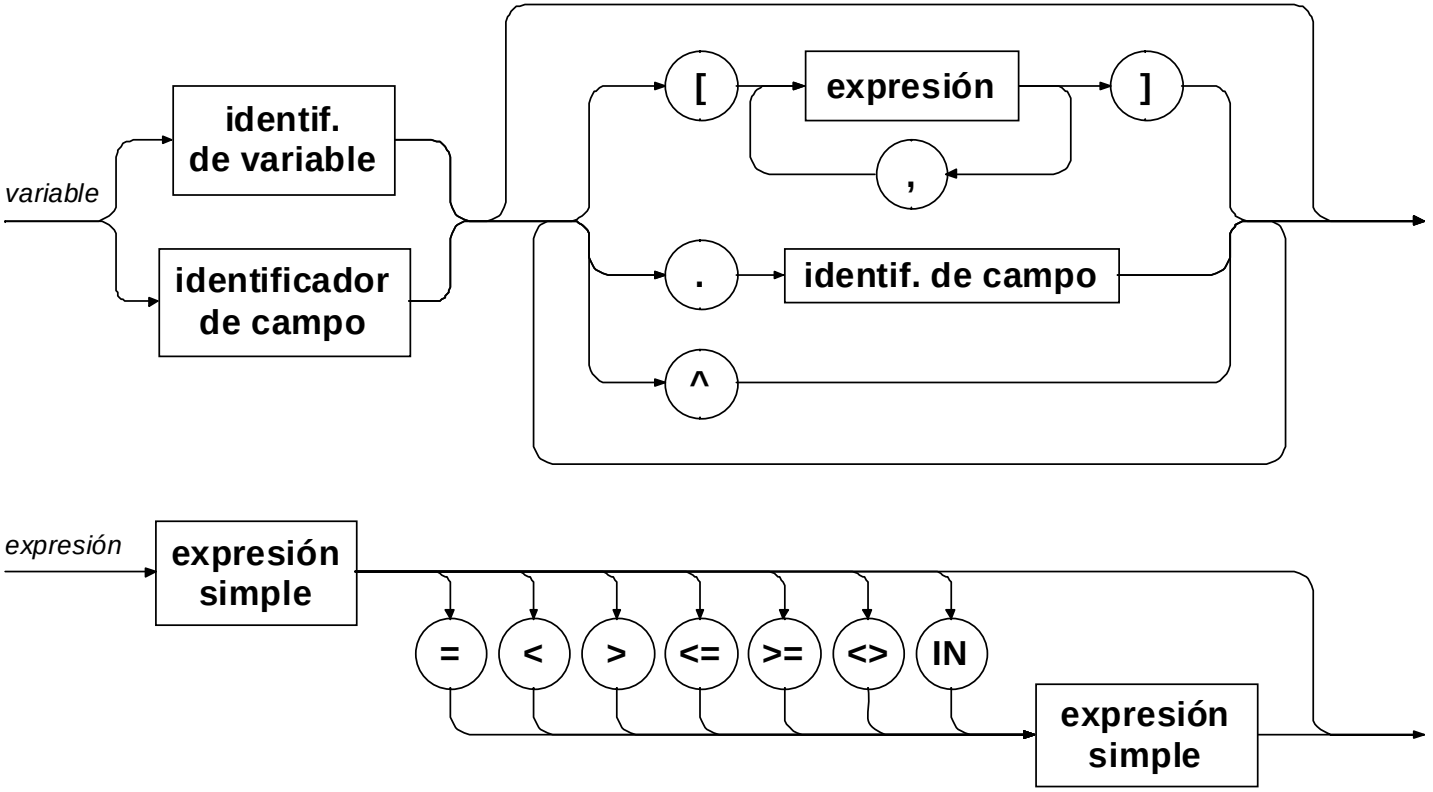
Probar que se trata de una gramática NO LL(1). Si es posible, realizar las transformaciones precisas para convertirla en LL(1) y, construyendo la tabla del análisis LL(1), demostrar que la gramática transformada es efectivamente de esta clase. Es necesario justificar las transformaciones que se realicen. Si no fuera posible transformarla en LL(1), explicar la razón

- **Ejercicio 2 (1.5 ptos.)**: Dada una secuencia ω determinales el algoritmo de análisis LL(1) emite la secuencia de producciones aplicadas para su derivación, o error si no es capaz de encontrarla. Vamos a denotar n el número de no terminales de la gramática, y $|\omega|$ la longitud de la secuencia de entrada. Se sabe que, en el caso de que el algoritmo dé respuesta afirmativa, el coste de ejecución del algoritmo es $\Omega(|\omega|)$, $O(n|\omega|)$. En el cálculo de este coste se han considerado como operaciones de coste constante las siguientes:
  - obtener la cima de la pila
  - apilar y desapilar símbolos gramaticales
  - obtener el siguiente terminal de la sencuencia de entrada
  - consultar el valor en la tabla correspondiente a un par (no terminal, terminal)
  - emitir una producción

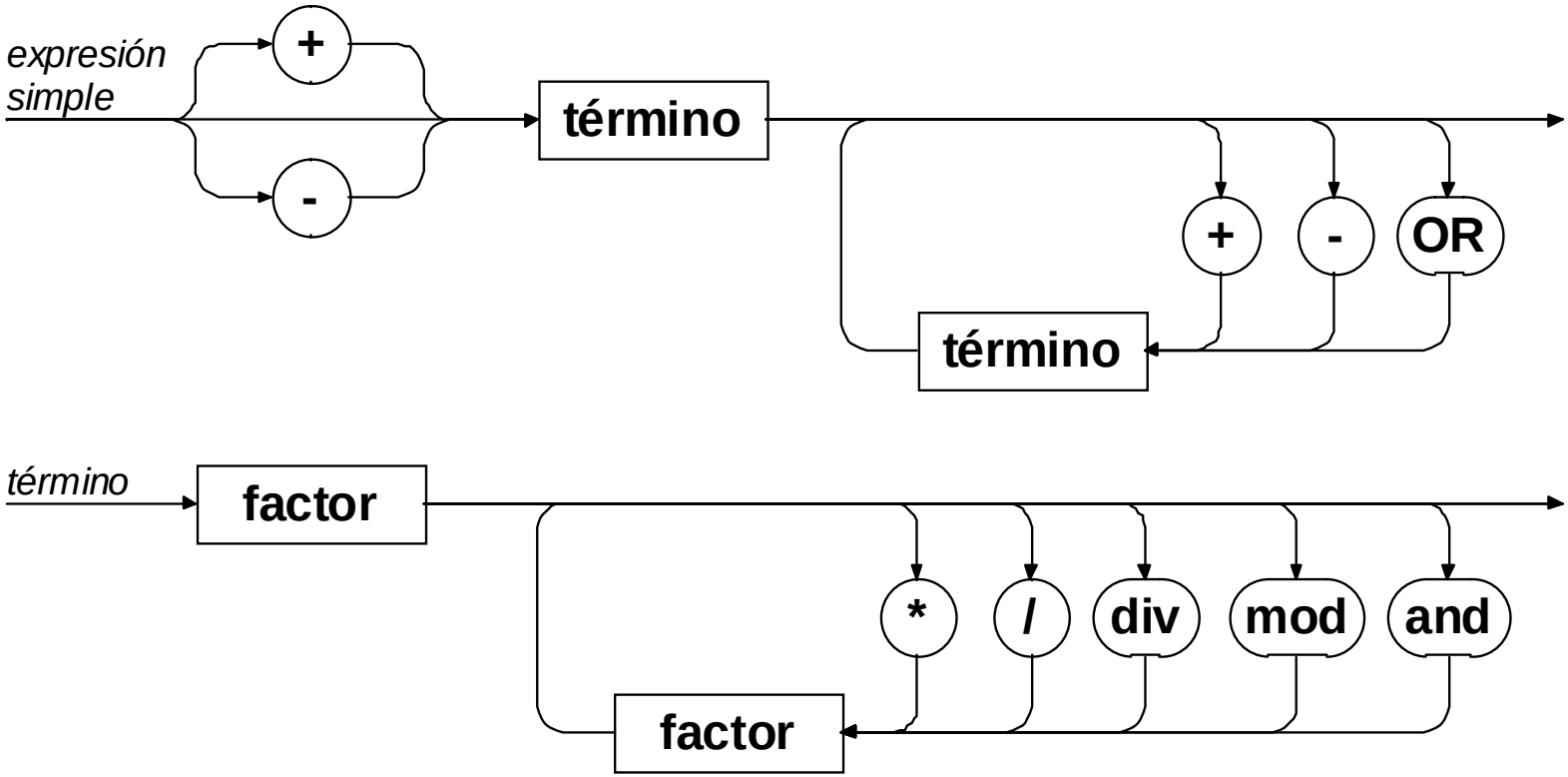
Ejercicios

- Consideremos ahora la siguiente gramática:
  - X_1 --> X_2 X_2 X_2
  - X_2 --> a_2 | X_3
  - X_3 --> a_3 | X_4
  - X_4 --> a_4 | X_5
  - ....
  - X_{n-1} --> a_{n-1} | X_n
  - X_n --> a_n
- El ejercicio pide escribir dos frases del lenguaje generado por la gramática anterior, w_1 y w_2, tal que los costes de ejecución del algoritmo de reconocimiento sean, respectivamente, del orden de $|w_1|$ y $n|w_2|$.
- ¿Cuál puede ser el coste del algoritmo, en el mejor y el peor caso, cuando se aplica a una frase que no es reconocida por el análisis LL(1)?
- **Nota**: en todos los casos la respuesta debe ser razonada.

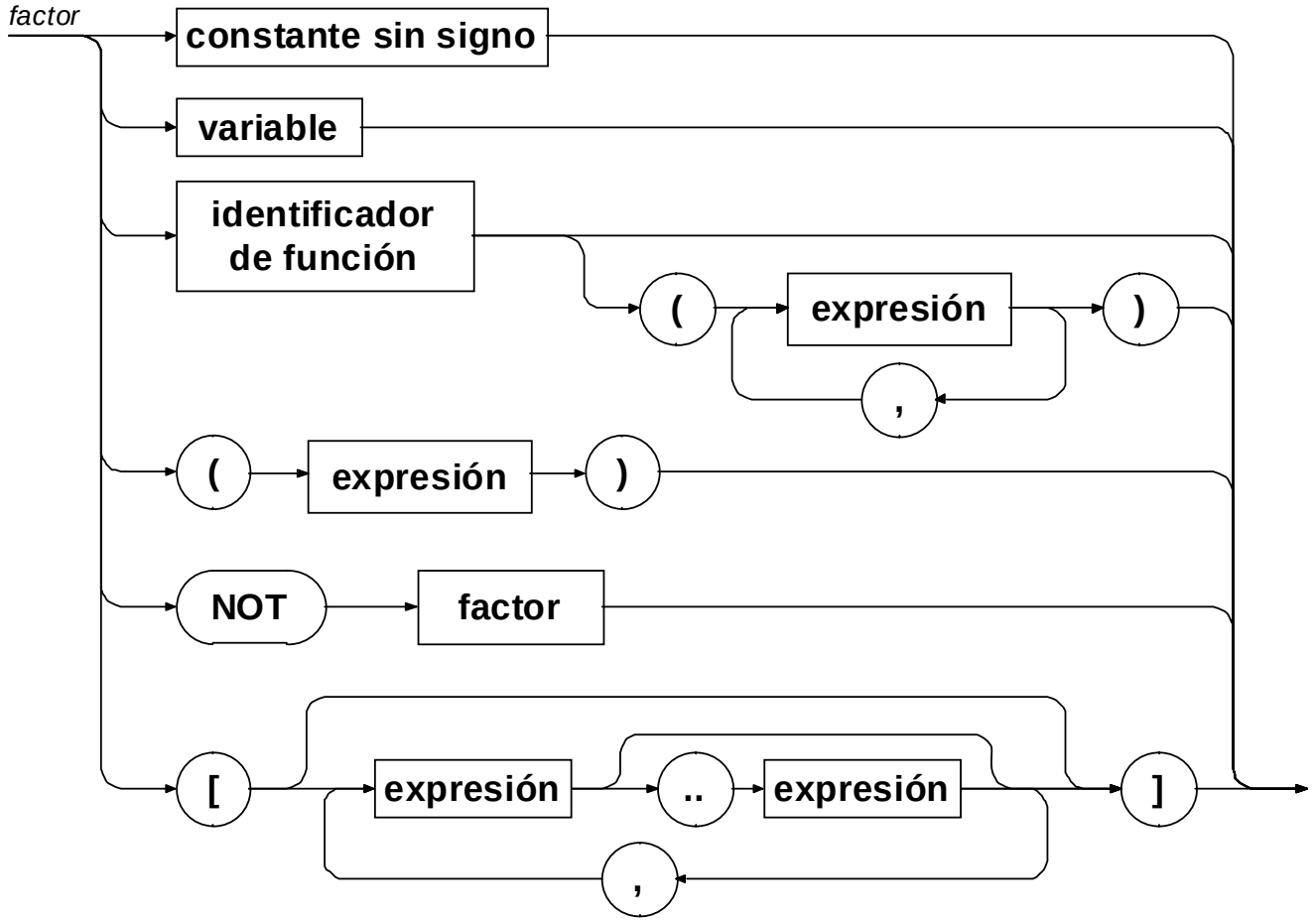
Sintaxis expresiones en Pascal

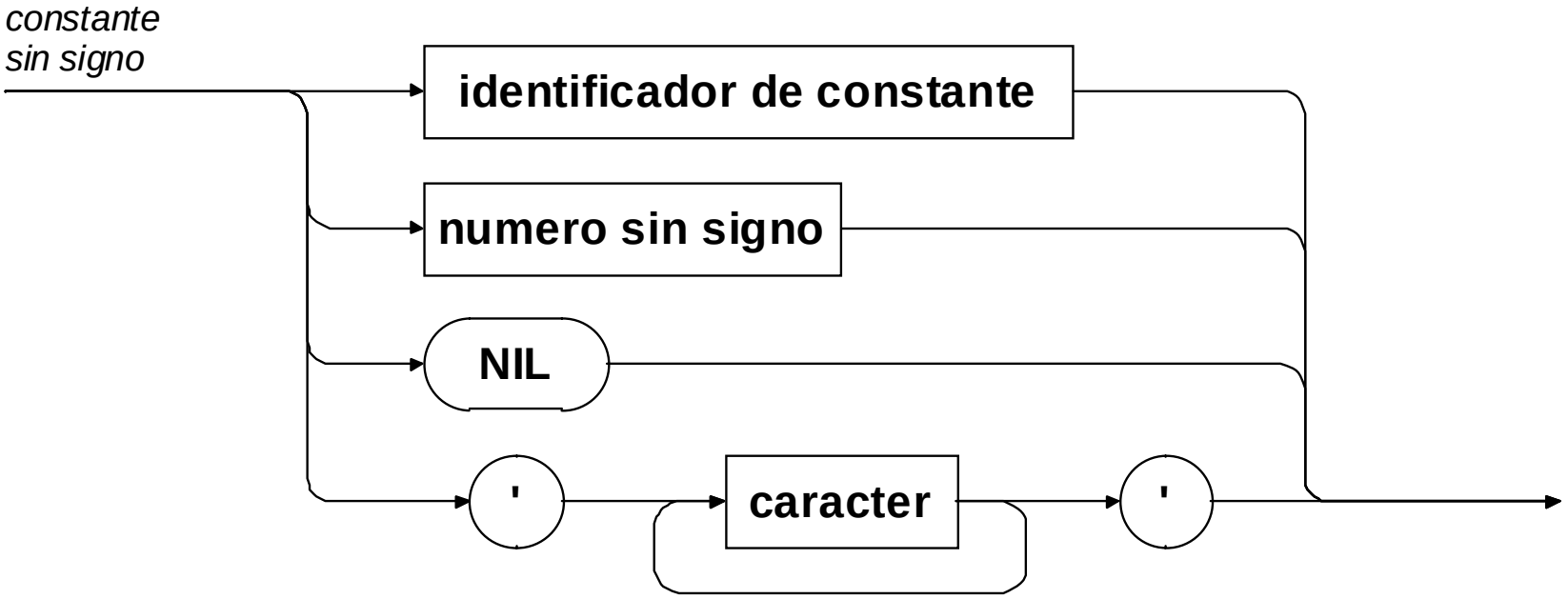


Sintaxis expresiones en Pascal



Sintaxis expresiones en Pascal




constante
sin signo
identificador de constante
numero sin signo
NIL
'
caracter
'

Sintaxis expresiones en Pascal

```
expr:
  expSimple
| expr opRel expSimple

expSimple:
  term
| expSimple opSum term

opRel:
  '='
| '<'
| '>'
| MEI
| MAI
| NI
```

```
opSum:
  '+'
| '-'
| OR

term:
  factor
| '+'  term
| '-'  term %prec '-'
| term opMul factor

opMul:
  '*'
| '/'
| DIV
| MOD
| AND
```

```
factor:
  constante
| IDENTIFICADOR
| IDENTIFICADOR '[' expr ']'
| IDENTIFICADOR  '(' listaActuales ')'
| '(' expr  ')'
| NOT factor %prec NOT

constante:
  CONSTENTERA
| CONSTBOOLEANA
| CONSTCADENA
| CONSTCARACTER
```

Sintaxis expresiones en Ada

[...]: 0 ó 1
{...}: 0 ó más

```
expression ::=
  relation {and relation}
| relation {and then relation}
| relation {or relation}
| relation {or else relation}
| relation {xor relation}

relation ::=
  simple_expression [relational_operator simple_expression]
| simple_expression [not] in range
| simple_expression [not] in subtype_mark

simple_expression ::=
  [unary_adding_operator] term {binary_adding_operator term}

term ::=
  factor {multiplying_operator factor}

factor ::=
  primary [** primary]
| abs primary
| not primary
```

Sintaxis expresiones en Ada

[...]: 0 ó 1
{...}: 0 ó más

```
primary ::=
  numeric_literal
| null
| string_literal
| aggregate
| name
| qualified_expression
| allocator
| (expression)

logical_operator ::= and | or | xor

relational_operator ::= = | /= | < | <= | > | >=

binary_adding_operator ::= + | - | &

unary_adding_operator ::= + | -

multiplying_operator ::= * | / | mod | rem

highest_precedence_operator ::= ** | abs | not
```