Universidad de Valladolid

Departamento de Informática

Teoría de autómatas y lenguajes formales . 2º I.T.Informática. Gestión.

Examen ordinario. 7 de junio de 2004. Cincuentenario de la muerte de Turing

1 **(1'75 p.)**

1. Es falso. Por ejemplo, si $L = \varepsilon$: $\varepsilon^* - \varepsilon = \emptyset \neq \varepsilon^+ = \varepsilon$
(Más general:
$L^* - \varepsilon$ no contiene la cadena vacía, pero L^+ puede contenerlo, exactamente cuando $\varepsilon \in L$.
Lo que siempre es cierto es que $L^* = L^+ \cup \varepsilon$, por definición.
Si $\varepsilon \in L$ entonces también $\varepsilon \in L^+$, luego $L^* = L^+ \cup \varepsilon = L^+$

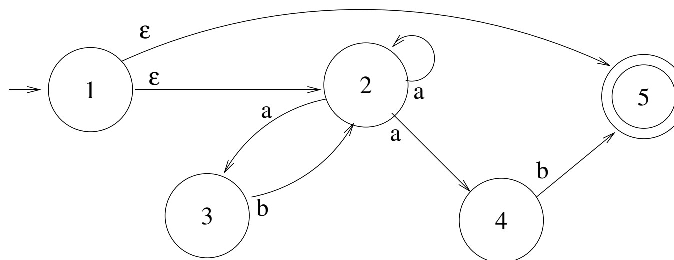
$$\text{Si } \varepsilon \in L \text{ entonces } L^* - \varepsilon \neq L^+$$

Si $\varepsilon \notin L$ entonces, en la expresión $L^* = \varepsilon \cup L^+$, la unión es disjunta y por lo tanto:

$$\text{Si } \varepsilon \notin L \text{ entonces } L^* - \varepsilon = L^+$$

)

2. En este caso sí se da la igualdad anterior: $(aa^*b)^* - \varepsilon = (aa^*b)^+$.
Por otra parte, la expresión $(ab\,|\,a)^*ab\,|\,\varepsilon$ es el lenguaje reconocido por el RF no determinista:



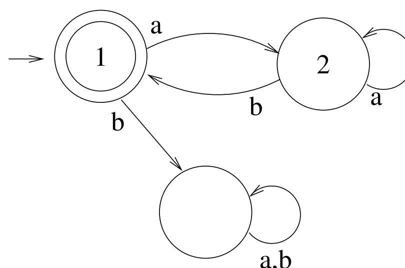
El algoritmo de determinación obtiene

	a	b	
$\rightarrow (A)$	B	C	$1,2,5$
B	B	D	$2,3,4$
C	C	C	
(D)	B	C	$2,5$

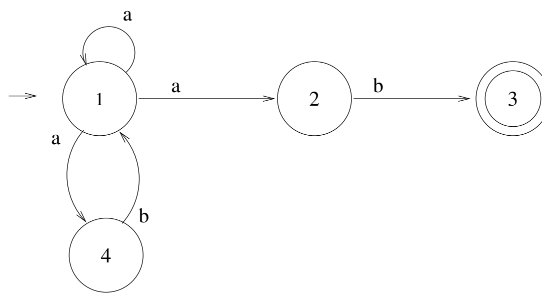
En el cálculo del RF mínimo se obtiene

$$\mathcal{P}_0 = \{\{A,D\}\,\{B,C\}\}$$
$$\mathcal{P}_1 = \{\{A,D\}\,\{B\}\,\{C\}\}$$

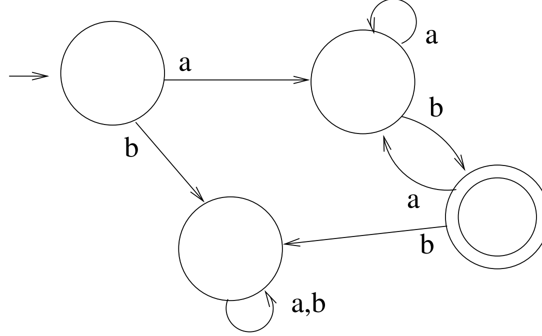
es decir:



del que claramente se obtiene la expresión para el lenguaje reconocido $(aa^*b)^*$
Por lo tanto $(ab\,|\,a)^*ab\,|\,\varepsilon = (aa^*b)^*$ y, al ser la unión disjunta, $(ab\,|\,a)^*ab = (aa^*b)^* - \varepsilon$
Otra forma de hacerlo sería calcular un RFN para $(ab\,|\,a)^*ab$:



y a partir de éste, el RFD mínimo:



del que se obtiene la expresión para el lenguaje reconocido $aa^*b(aa^*b)^*$, es decir $(aa^*b)^* - \varepsilon$

Aún otra forma:

$(aa^*b)^* - \varepsilon = (aa^*b)^+ = (\text{def}) = (aa^*b)^*aa^*b = (11) = (a^*ab)^*a^*ab = (17) = (a|ab)^*ab$

Los números de las igualdades se refieren a las propiedades vistas en clase:

$11 : \alpha^*\alpha = \alpha\alpha^*$

$17 : (\alpha^*\beta)^*\alpha^* = (\alpha|\beta)^*$

3. Ahora $\alpha^* = a^*$ y
   $\gamma\beta = \gamma \quad \forall\gamma$, por lo tanto
   $(\alpha\alpha^*\beta)^* = ((\varepsilon|a)a^*)^* - \varepsilon = (a^*|aa^*)^* - \varepsilon = (a^*)^* - \varepsilon = a^* - \varepsilon = \boxed{a^+}$
   mientras que
   $(\alpha\beta \,|\, \alpha)^*\alpha\beta = (\alpha \,|\, \alpha)^*\alpha = (\alpha)^*\alpha = a^*(\varepsilon \,|\, a) = a^* \,|\, a^*a = \boxed{a^*}$
   luego no coinciden.

2 **(1 p.)** Si lo fuera, cumpliría el lema de bombeo de los l. i. c. Sea N una constante para este lema. Sea P un número primo estrictamente mayor que N (siempre lo hay, porque hay infinitos primos, que evidentemente no pueden ser todos menores o iguales que N, y además, como $N > 0$, $P > 1$)

Sea $z = a^P$. Para cualquier descomposición de z en $xyuvw$ con $|yv| > 0$, evidentemente $y = a^r$, $v = a^s$ con $q = r + s > 0$, de forma que x, u y w estarán constituídas por el resto de las aes. Por lo tanto, para cualquier $i >= 0$, $|xy^iuv^iw| = |xyuvw| + |y^{i-1}v^{i-1}| = |xyuvw| + |(yv)^{i-1}| = P + (i-1)q$

Cumplir el lema de bombeo se traduce entonces en que, para cierto $q > 0$, y para todo $i >= 0$ se verifica que $P + (i-1)q$ es primo, lo que claramente no es cierto: por ejemplo, para $i = P + 1$, $P + (i-1)q = P + Pq = P(1+q)$ no es primo, ya que es el producto de P y $q+1$, ninguno de los cuales es 1.

3 **(1'5 p.)** 1. Si $P = (\Sigma, \Gamma, Q_P, A_0, q_1, f_P, F_P)$ es el autómata a pila determinista en cuestión, y $M = (\Sigma, Q_R, p_1, f_R, F_R))$ es un autómata finito determinista cuyo lenguaje reconocido es R, la función de transición del autómata a pila que se construye en la demostración de que la intersección de un independiente de contexto y un regular es independiente de contexto, se define de la siguiente manera:

$$f_{PM}([q,p], a, A) := \begin{cases} (\; [f_{P|1}(q,a,A) \; , \; f_q(p,a)] \; , \; f_{P|2}(q,a,A) \;) & \text{si } a \in \Sigma \\ (\; [f_{P|1}(q,\varepsilon,A) \; , \; p] \; , \; f_{P|2}(q,\varepsilon,A) \;) & \text{si } a = \varepsilon \end{cases}$$

siendo $f_{P|1} y f_{P|2}$ las componentes primera y segunda respectivamente de la función f_P.

Explicado de otra manera: el autómata que se construye (PM) para reconocer la intersección, tiene por estados el producto cartesiano de los conjuntos de estados de P y M. Su función de transición mantiene, en la primera coordenada, los estados a los que transita cada uno de ellos (la segunda coordenada se reserva para el tratamiento de la -única- pila).

El reconocedor para el lenguaje regular no tiene arcos ε, pero el autómata a pila puede tenerlos. En ese caso, la parte del autómata finito no cambia de estado:

Si $f_P(q,a,A) = \{(q',\alpha)\}$ y $f_M(p,a) = p'$, se define $f_{PM}([q,p],a,A) = \{([q',p'],\alpha)\}$
y si $f_P(q,\varepsilon,A) = \{(q',\alpha)\}$, se define $f_{PM}([q,p],\varepsilon,A) = \{([q',p],\alpha)\}$

Por lo tanto, es claro que

- $\forall([q,p] \in Q_{PM}, A \in \Gamma, a \in \Sigma_E \cup \{\varepsilon\}) \qquad \#f_{PM}(q,a,A) \leq 1$
  (puesto que tanto f_P como f_R definen una única imagen por cada uno de los elementos de los conjuntos origen)
- y $\forall([q,p] \in Q_{PM}, A \in \Gamma, a \in \Sigma_E)$
  $f_{PM}([q,p],\varepsilon,A) \neq \emptyset \Rightarrow f_P(q,\varepsilon,A) \neq \emptyset \Rightarrow f_P(q,a,A) = \emptyset \Rightarrow f_{PM}([q,p],a,A) = \emptyset$

(dada la definición de f_{PM} y el hecho de que P es determinista)

Por lo tanto se cumplen las condiciones de determinismo de f_{PM}

Es FALSO:

- que la intersección de dos lenguajes independientes de contexto lo sea (en general)
- que se pueda construir, en general, un autómata a pila como intersección de dos autómatas a pila (o sea, que se puedan reducir dos pilas a una)
- que un lenguaje reconocible por estado final por un autómata a pila determinista sea regular (ver uno de los primeros ejemplos vistos en clase)
- que cualquier autómata a pila pueda convertirse en determinista

2. En las condiciones del lenguaje dado, si $|w|_a$ es par, entonces $|wcw^R|_a$ es múltiplo de 4. Recíprocamente, como $|wcw^R|_a = 2|w|_a$, si $|wcw^R|_a$ es múltiplo de 4 entonces $|w|_a$ es par.

   El lenguaje es independiente de contexto, puesto que es la intersección de
   $L_i = \{wcw^R \,/\, w \in (a|b)^*\}$, independiente de contexto y
   $L_R = \{x \in (a|b|c)^* \,/\, |x|_a =$ múltiplo de 4 $\}$, regular ($= (\beta^*a\beta^*a\beta^*a\beta^*a\beta^*)^*$ siendo $\beta = b|c$)

   Una gramática que lo genera es, por ejemplo

$$\begin{cases} S \rightarrow & aRa \mid bSb \mid c \\ R \rightarrow & aSa \mid bRb \end{cases}$$

   (R lleva la cuenta de número impar de aes en cada mitad, y S de número par).

   Una forma de obtener otra gramática sería partir de la intersección de un reconocedor finito para L_R y un autómata a pila para L_i que lo reconociese por estado final, y aplicar el algoritmo de análisis correspondiente para llegar a una gramática:

```
Construir a partir de {S → aSa | bSb | c} un AP que reconoce por vaciado de pila
          (vía directa o mediante FNG)
Construir a partir del anterior un AP que reconozca por estado final
Construir un RFD para L_R, a partir de su expresión regular, por ejemplo
Construir el AP intersección de los dos anteriores
Construir a partir del anterior un AP que reconozca por vaciado de pila
Obtener a partir del anterior una gramática independiente de contexto
```

   (los algoritmos para los 6 pasos se conocen).

   Está mal:

   - demostrar que el lenguaje es independiente de contexto probando que verifica el lema de bombeo de los l.i.c.

- intentar demostrar que no lo es probando que no verifica el lema de bombeo (porque lo verifica, porque de hecho es independiente de contexto. Constante del lema: cualquiera mayor o igual que 5. Las cadenas del lenguaje más largas tienen una parte central *aacaa*, *bacab* ó *bcb*. En el primer caso se bombean $y = aa$ y $v = aa$. En los otros dos, las bes).

5 (1'5 p.) 1. Una gramática cualquiera vendrá codificada por la sucesión de las reglas:

$r(cr)^*$ si se supone que hay al menos una regla, ó $\varepsilon | r(cr)^*$ si no

(no de forma única, puesto que las reglas pueden darse en distinto orden)

Si la gramática es independiente de contexto, todas sus reglas lo son.

Las reglas independientes de contexto son de la forma *auxiliar* → *cadena de símbolos*, donde la cadena del consecuente puede ser o no vacía. Si lo es, se codificará por e, y si no, será una cadena no nula sobre $\{S, A, B, a, b\}$.

Por lo tanto, $r = (S|A|B)f((S|A|B|a|b)^+|e)$

La expresión sustituyendo resulta:

$$\varepsilon|((S|A|B)f((S|A|B|a|b)^+|e)(c((S|A|B)f((S|A|B|a|b)^+|e))^*$$

La descripción se entiende mejor mediante la definición regular siguiente:

auxiliar	→	$S \mid A \mid B$
terminal	→	$a \mid b$
símbolo	→	*auxiliar* | *terminal*
consecuente	→	e | *símbolo*$^+$
regla	→	*auxiliar* f *consecuente*
gramática	→	ε | *regla* (c *regla*)*

2. Un lenguaje es recursivamente numerable si y sólo si existe una máquina de Turing (o codificación en Pascal) cuyo lenguaje reconocido sea el lenguaje en cuestión (se pare o no ante las cadenas que no son del lenguaje).

   Se puede construir tal algoritmo, basándose en las siguientes propiedades:

   a) Si la gramática tiene una regla $A \to A$, es ambigua

   b) Si la gramática es $LL(1)$, no es ambigua

   c) Si la gramática es $LALR(1)$, no es ambigua

   d) Si ninguno de los algoritmos anteriores determinó la ambigüedad de la gramática, se pueden generar sucesivamente las derivaciones más a la izquierda (o sea, árboles de derivación) de longitudes cada vez mayores, e ir anotando las cadenas obtenidas. Si una de ellas aparece por segunda vez, la gramática será ambigua. Si no, el proceso no terminará. Pero desde luego, si la gramática es ambigua, antes o después se detectará que lo es.

   (Bastaría con el proceso *d* para justificar que el lenguaje es recursivamente numerable). Un algoritmo basado en estos hechos sería como sigue:

```
0. Leer G (independiente de contexto)
1. Si existe una regla A -> A, escribir "Es ambigua" y Parar.
2. Calcular la TASP.
3. Si en ninguna casilla hay más de una regla, escribir "No es ambigua" y Parar.
4. Reformularla como entrada para YACC, y compilar con YACC.
5. Si Yacc no detecta conflictos, escribir "No es ambigua" y Parar.
6. Numerar las reglas : r_1, r_2, ... , r_n.
7. EsAmbigua := no sé
8. FS := {S}; SENT := conjunto vacío
9. repetir
    FS' := conjunto vacío (FS'=formas sentenciales en un paso más)
    para cada fs de FS hacer
       para i desde 1 hasta n hacer
          aplicar la regla r_i al símbolo de más a la
```

```
                izquierda de fs (si se puede)
         si el resultado está constituído por terminales
            si ya estaba en el conjunto de sentencias
                EsAmbigua := Sí
            si no
                añadirlo a SENT
         si no
            añadirlo a FS'
      FS := FS'
   hasta EsAmbigua = Sí
10.  Escribir "Es ambigua"
```

(No serían necesarios los pasos 1 a 5)

6 **(1'75 p.)**

1. La parte del algoritmo para calcular la TASP que involucra reglas con consecuentes no anulables está realizada correctamente.

   Si se sabe que la gramática sólo tiene un símbolo anulable, sólo hay una regla $N \to \varepsilon$ y ninguna otra tiene consecuentes anulables. Por lo tanto, el algoritmo debería calcular los siguientes de tal N y terminar colocando la regla $N \to \varepsilon$ en las casillas correspondientes a dichos siguientes de la fila N. Como se sabe que la gramática es $LL(1)$, ninguno de estos lugares estará ya ocupado.

   Nuestro estudiante, sencillamente, coloca más reglas de las estrictamente necesarias.

   Por lo tanto, todas las cadenas que el analizador correcto acepta, siguen un recorrido de la TASP que se puede realizar en su solución, así que $L(G) \subseteq L$, si L es el lenguaje que reconoce el analizador de nuestro estudiante.

   Una reflexión más profunda da lugar a deducir que tampoco en L hay más cadenas que en $L(G)$: Supongamos que en un determinado momento del análisis descendente se accede por primera vez a una casilla que contiene una regla $N \to \varepsilon$ mal colocada:

   $$\text{TASP}[N, a] \text{ con } a \notin \text{Primeros}(N) \text{ y } a \notin \text{Siguientes}(N).$$

   Se habrá realizado hasta ese momento un recorrido de un árbol de derivación correcto dando una forma sentencial izquierda:

   $$S \Rightarrow^* xN\beta$$

   donde todos los terminales de x ya están pareados y se sabe que

   $$a \notin \text{Primeros}(N), \text{ y } a \notin \text{Primeros}(\beta)$$

   - Si β no es anulable, la aplicación de $N \to \varepsilon$ en este caso, dará lugar a que en un momento posterior, el primer símbolo distinto de N de β o bien es un terminal $b \neq a$ que se intenta parear (sin éxito) con a, o bien es un auxiliar B que se enfrenta a a encontrando una casilla vacía. Se detecta el error en ese momento (algo más tarde de lo que haría el analizador predictivo estándar).
   - β es anulable ($\beta = NN \ldots N$ o directamente ε). Ahora, si a no es \$, por más que se apliquen reglas $N \to \varepsilon$, nunca se parea a, el análisis no consume la cadena de entrada, y detecta error. Y $a \neq \$$, porque si es posible la derivación $S \Rightarrow^* xNN \ldots N$ entonces \$ está en Siguientes(N).

   Por lo tanto, el acceso a una casilla de la TASP con una regla $N \to \varepsilon$ colocada incorrectamente, dará lugar también a la detección de error (aunque algo más tarde): no se analizan más cadenas de las que se debe.

2. Claramente, $L_N = b^*$ y a partir de S, sin usar N se genera exactamente $\{a^n N b^n \,/\, n \geq 0\}$, luego

   $$L(G) = \{a^n b^m b^n \,/\, n, m \geq 0\} == \{a^n b^m \,/\, m \geq n \geq 0\}$$

   Primeros(S) $= \{a, b, \varepsilon\}$ y Primeros(N) $= \{b, \varepsilon\}$. La tabla que construye nuestro estudiante es

	a	b	\$
S	$S \to aSb$	$S \to N$	
N	$N \to \varepsilon$	$N \to bN$	$N \to \varepsilon$

Los análisis que realiza son los siguientes:

pila	entrada	
$\$S$	$\$$	error

pila	entrada	
$\$S$	$ab\$$	$S \to aSb$
$\$bSa$	$ab\$$	parea
$\$bS$	$b\$$	$S \to N$
$\$bN$	$b\$$	$N \to bN$
$\$bNb$	$b\$$	parea
$\$bN$	$\$$	$N \to \varepsilon$
$\$b$	$\$$	error

pila	entrada	
$\$S$	$abb\$$	$S \to aSb$
$\$bSa$	$abb\$$	parea
$\$bS$	$bb\$$	$S \to N$
$\$bN$	$bb\$$	$N \to bN$
$\$bNb$	$bb\$$	parea
$\$bN$	$b\$$	$N \to bN$
$\$bNb$	$b\$$	parea
$\$bN$	$\$$	$N \to \varepsilon$
$\$b$	$\$$	error

pila	entrada	
$\$S$	$aab\$$	$S \to aSb$
$\$bSa$	$aab\$$	parea
$\$bS$	$ab\$$	$S \to aSb$
$\$bbSa$	$ab\$$	parea
$\$bbS$	$b\$$	$S \to N$
$\$bbN$	$b\$$	$N \to bN$
$\$bbNb$	$b\$$	parea
$\$bbN$	$\$$	$N \to \varepsilon$
$\$bb$	$\$$	error

Por lo que se ve, no reconoce lo que debe (por ejemplo, ε ni ab ni abb).
Veamos como debe ser una cadena reconocida:
Si la cadena empieza por a, el análisis comienza

pila	entrada	
$\$S$	$a\alpha\$$	$S \to aSb$
$\$bSa$	$a\alpha\$$	parea
$\$bS$	$\alpha\$$	...

S debe salir de la pila: por lo tanto, después, posiblemente, de varias aplicaciones de $S \to aSb$, siendo $\alpha = a^n\alpha_1$ con $n \geq 0$, se llegará a una aplicación de $S \to N$ porque se encuentra una b en la entrada: $\alpha_1 = b^k\beta$ con $k \geq 1$

pila	entrada	
$\$S$	$a\alpha\$$	$S \to aSb$
$\$bSa$	$a\alpha\$$	parea
$\$bS$	$\alpha\$$	...
...		
$\$bb^nS$	$\alpha_1\$$	=
$\$bb^nS$	$b^k\beta\$$	$S \to N$
$\$bb^nN$	$bb^{k-1}\beta\$$	

Entonces, se aplicará por fuerza la regla $N \to bN$, se pareará b y se continuará aplicando $N \to bN$ mientras haya bes:

pila	entrada	
$\$S$	$a\alpha\$$	$S \to aSb$
$\$bSa$	$a\alpha\$$	parea
$\$bS$	$\alpha\$$	...
...		
$\$bb^nS$	$\alpha_1\$$	=
$\$bb^nS$	$b^k\beta\$$	$S \to N$
$\$bb^nN$	$bb^{k-1}\beta\$$	$N \to bN$
$\$bb^nNb$	$bb^{k-1}\beta\$$	parea
$\$bb^nN$	$b^{k-1}\beta\$$	...
...		
$\$bb^nN$	$\beta\$$	

La única manera de sacar N de la pila es encontrar ahora a ó \$:
Si $\beta = a\beta_1$, el análisis prosigue

pila	entrada	
...		
$\$bb^nN$	$a\beta_1\$$	$N \to \varepsilon$
$\$bb^n$	$a\beta_1\$$	error

O bien, si $\beta = \varepsilon$:

pila	entrada	
...		
$\$bb^nN$	\$	$N \to \varepsilon$
$\$bb^n$	\$	error

Por lo tanto no se acepta ninguna cadena que empiece por a

Si la cadena empieza por b, el análisis comienza

pila	entrada	
$\$S$	$b\beta\$$	$S \to N$
$\$N$	$b\beta\$$	$N \to bN$
$\$Nb$	$b\beta\$$	parea
$\$N$	$\beta\$$	

El análisis puede continuar leyendo y pareando bes sin que N salga de la pila. Para que salga, debe encontrarse a ó \$:

pila	entrada	
$\$S$	$bb^n\beta\$$	$S \to N$
$\$N$	$bb^n\beta\$$	$N \to bN$
$\$Nb$	$bb^n\beta\$$	parea
...		
$\$N$	$a\beta_1\$$	$N \to \varepsilon$
\$	$a\beta_1\$$	error

ó

pila	entrada	
$\$S$	$bb^n\$$	$S \to N$
$\$N$	$bb^n\$$	$N \to bN$
$\$Nb$	$bb^n\$$	parea
...		
$\$N$	\$	$N \to \varepsilon$
\$	\$	aceptar

Por lo tanto se aceptan solamente cadenas de b^+

3. Para calcula la TASP correcta hay que calcular los siguientes:
   Siguientes(S)= Siguientes(N) $= \{\$, b\}$
   y la tabla es:

	a	b	\$
S	$S \to aSb$	$S \to N$	$S \to N$
N		$N \to bN; N \to \varepsilon$	$N \to \varepsilon$

   La gramática no es $LL(1)$.

4. Las derivaciones más a la derecha , marcando los consecuentes de los pivotes, son:

$$
\begin{aligned}
S &\Rightarrow \underline{N} \Rightarrow \underline{\varepsilon} \\
S &\Rightarrow \underline{N} \Rightarrow \underline{bN} \\
&\Rightarrow b\underline{bN} \Rightarrow^* b^k\underline{bN} \Rightarrow b^{k+1}\underline{\varepsilon}i \qquad K \geq 0 \\
S &\Rightarrow \underline{aSb} \\
&\Rightarrow a\underline{aSb}b \Rightarrow^* a^n\underline{aSb}b^n \qquad n \geq 0 \\
&\Rightarrow a^{n+1}\underline{N}b^{n+1} \Rightarrow a^{n+1}\underline{\varepsilon}b^{n+1} \\
S &\Rightarrow\Rightarrow^* a^n\underline{aSb}b^n \qquad n \geq 0 \\
&\Rightarrow a^{n+1}\underline{N}b^{n+1} \Rightarrow a^{n+1}\underline{bN}b^{n+1} \\
&\Rightarrow a^{n+1}b\underline{bN}b^{n+1} \Rightarrow^* a^nb^k\underline{bN}b^n \Rightarrow a^nb^{k+1}\underline{\varepsilon}b^n \quad k \geq 0
\end{aligned}
$$

   Es decir, las formas sentenciales derechas son

   - $a^nb^k\underline{\varepsilon}b^n$ con $k \geq 0$ y $n \geq 0$

- $a^n \underline{N} b^N$ con $n \geq 0$
- $a^n b^k \underline{bN} b^n$ con $k \geq 0$ y $n \geq 0$
- $a^n b^k \underline{aSb} b^n$ con $k \geq 0$, $n \geq 0$

o, un poco más desglosadas, si se entiende mejor:

a) $\underline{\varepsilon}$
b) $b^k \underline{\varepsilon}$ con $k > 0$
c) $a^n \underline{\varepsilon} b^n$ con $n > 0$
d) $a^n b^k \underline{\varepsilon} b^n$ con $k > 0$, $n > 0$
e) $\underline{N}$
f) $b^k \underline{bN}$ con $k \geq 0$
g) $a^n \underline{N} b^n$ con $n > 0$
h) $a^n b^k \underline{bN} b^n$ con $n > 0$, $k \geq 0$
i) $a^n \underline{aSb} b^n$ con $n \geq 0$
j) $a^n b^k \underline{aSb} b^n$ con $k > 0$, $n > 0$

G no puede ser $LALR(1)$, porque no hay manera de localizar el pivote en los casos c y d leyendo de izquierda a derecha: el pivote es ε (con $N \rightarrow \varepsilon$), en la posición que deje a la derecha tantas bes como aes haya al principio.

7 **(1'5 p.)** Una solución: Fuente YACC

```
%{
#include <string.h>
%}
%union {struct {
            char tipo;
            float val;
            char cad [256];
            };
       };
%token ID DATO ASIGN
%left '+' '-' OR
%left '*' NOT AND
%%
S      : ID ASIGN exp ';'
            {if ($1.tipo == $3.tipo) {
                switch ($1.tipo) {
                    case 'i': printf ("%d\n", (int)$3.val); break;
                    case 'r': printf ("%f\n", $3.val); break;
                    case 'b':   if ($3.val==1) printf ("TRUE\n");
                                else           printf ("FALSE\n");
                                break;
                    case 's' : printf ("%s\n", $3.cad); break;
                }
            } else printf ("error de tipo");
            }
       ;
exp    : exp '+' exp
            {if ($1.tipo == $3.tipo) {
                switch ($1.tipo) {
                    case 'i' : $$.tipo = 'i';
                               $$.val = $1.val + $3.val; break;
                    case 'r' : $$.tipo = 'r';
                               $$.val = $1.val + $3.val; break;
```

```
                    case 'b' : printf ("no hay suma entre boolean\n");
                               break;
                    case 's' : $$.tipo = 's';
                               strcat($1.cad, $3.cad);
                               strcpy($$.cad, $1.cad); break;
                }
        } else printf ("error de tipo\n");
        }
    | exp '-' exp
        {if ($1.tipo == $3.tipo) {
            if ($1.tipo=='i' || $1.tipo == 'r') {
                $$.tipo = $1.tipo;
                $$.val = $1.val - $3.val;
            } else printf ("no se pueden restar cadenas ni boolean\n");
        } else printf ("error de tipo\n");
        }
    | '-' exp %prec '*'
        {if ($2.tipo == 'i' || $2.tipo == 'r'){
            $$.tipo = $2.tipo;
            $$.val = - $2.val;
        } else printf ("no se puede cambiar de signo\n");
        }
    | exp '*' exp
        {if ($1.tipo == $3.tipo) {
            if ($1.tipo=='i' || $1.tipo == 'r') {
                $$.tipo = $1.tipo;
                $$.val = $1.val * $3.val;
            } else printf ("no se pueden multiplicar cadenas o boolean\n");
        } else printf ("error de tipo\n");
        }
    |'(' exp ')'
        { $1.tipo = $2.tipo;
              if ($2.tipo=='s') strcpy ($$.cad, $2.cad);
              else $$.val = $2.val;
        }
    |exp AND exp
        {if ($1.tipo == $3.tipo) {
            if ($1.tipo=='b') {
                $$.tipo = 'b';
                $$.val = $1.val * $3.val;
            } else printf ("AND solamente entre boolean\n");
        } else printf ("error de tipo\n");
        }
    | exp OR  exp
        {if ($1.tipo == $3.tipo) {
            if ($1.tipo == 'b') {
                $$.tipo = 'b';
                $$.val = $1.val + $3.val;
                if ($$.val>1) $$.val = 1;
            } else printf ("OR solamente entre boolean\n");
        } else printf ("error de tipo\n");
        }
    | NOT exp
        { if ($2.tipo == 'b') {
```

```
                    $$.tipo = 'b';
                    if ($2.val == 1) $$.val = 0;
                    else $$.val = 1;
                } else printf ("NOT solamente sobre boolean\n"); }
        | DATO
                { $$.tipo = $1.tipo;
                  if ($1.tipo == 's') strcpy ($$.cad, $1.cad);
                  else $$.val = $1.val; }
        ;
%%
main() { yyparse(); }
```

Fuente LEX

```
%{
#include "y.tab.h"
%}
dig     [0-9]
id      [a-zA-Z0-9]*
%s      cadena
term    [;()*+-]
%%
[ \t]+                  ;
":="                    return ASIGN;
{term}                  return yytext[0];
[-]?{dig}*              { yylval.tipo = 'i';
                          yylval.val = atoi(yytext);
                          return DATO; }
[-]?{dig}*"."{dig}*     { yylval.tipo = 'r';
                          yylval.val =  atof(yytext);
                          return DATO; }
[aA][nN][dD]            return AND;
[oO][rR]                return OR;
[nN][oO][tT]            return NOT;
[fF][aA][lL][sS][eE]    { yylval.tipo = 'b';
                          yylval.val = 0;
                          return DATO; }
[tT][rR][uU][eE]        { yylval.tipo = 'b';
                          yylval.val = 1;
                          return DATO; }
[sS]{id}                { yylval.tipo = 's';
                          return ID; }
[rR]{id}                { yylval.tipo = 'r';
                          return ID; }
[bB]{id}                { yylval.tipo = 'b';
                          return ID; }
[iI]{id}                { yylval.tipo = 'i';
                          return ID; }
<cadena>[^']+           { yylval.tipo = 's';
                          strcpy (yylval.cad, yytext);
                          return DATO; }
<cadena>[']             BEGIN INITIAL;
"'"                     BEGIN cadena;
\n                      ;
.               printf ("Error : %c inesperado\n", yytext[0]);
```