Teoría de Lenguajes
Guía 8 Resuelta

Sebastián Taboh

2 de junio de 2018

Este apunte contiene soluciones a ejercicios de la octava guía de ejercicios del 1er cuatrimestre de 2017 de la materia Teoría de Lenguajes, que está dedicada a Parsers Descendentes, así como también extractos de las clases prácticas dadas en ese cuatrimestre.

Este documento fue creado cuando cursé en el 2017, en caso de encontrar errores, por favor comunicarlo por mail a sebi_282@hotmail.com.

Índice

1. Parsers recursivos descendentes predictivos

Vamos a hacer un repaso sobre este tipo de parsers.

- Parsers descendentes (parten del símbolo distinguido)
- Utilizan la información del próximo token (i.e., token corriente, o `tc`) y los SD's de cada regla
- Con eso,
  - se crea un procedimiento por cada no terminal, siendo el procedimiento principal (`Main`) el correspondiente al símbolo distinguido
  - dentro de cada procedimiento, se invocan los procedimientos de los no terminales incluidos, se busca los terminales esperados (`match`), o no se hace nada (si es un λ),
  - si hay más de regla para un mismo no terminal, se agregan condicionales para ejecutar sólo la regla que contenga al token actual (`tc`) en sus SD's.
    Se agrega un `else` final con `error()`, para cubrir el caso en que el `tc` no está en los SD de ninguna de las reglas posibles.

Vamos a definir dos funciones: *Primeros* y *Siguientes*

$$Primeros(\alpha) : (V_N \cup V_T)^* \to \mathcal{P}(V_T)$$

$$Primeros(\alpha) = \{t \in V_T \mid \alpha \overset{*}{\Rightarrow} t\beta\}$$

$$Siguientes(N) : V_N \to \mathcal{P}(V_T)$$

$$Siguientes(N) = \{t \in V_T \mid S\$ \overset{*}{\Rightarrow} ...Nt...\}$$

Con eso creamos la función de Símbolos Directrices (SD):

$$SD(A \to \beta) = \begin{cases} Primeros(\beta) & \text{si } \beta \text{ no anulable } (\beta \not\Rightarrow^* \lambda) \\ Primeros(\beta) \cup Siguientes(A) & \text{si } \beta \text{ anulable } (\beta \Rightarrow^* \lambda) \end{cases}$$

1.1. Algunos problemas y cómo resolverlos

- Ambigüedad por SDs no disjuntos: factorización a la izquierda (*left-factorization*)
  Si tenemos producciones de la forma
  $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \delta_1 \mid \delta_2 \mid ... \mid \delta_k$ donde $\alpha \neq \lambda$, las reemplazamos por

  $$A \to \alpha A' \mid \delta_1 \mid \delta_2 \mid ... \mid \delta_k$$

  $$A' \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

- Recursión a izquierda
  - eliminación de la recursión inmediata
    Si tenemos producciones de la forma
    $A \to A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_k$ las reemplazamos por

    $$A \to \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_k A'$$

    $$A' \to \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \lambda$$

- eliminación de la recursión no inmediata
  Tenemos el siguiente algoritmo.
  - Numerar los no terminales $A_1, ..., A_n$
  - Para $i \leftarrow 1...n$
    - Para $j \leftarrow 1...i-1$
      - Reemplazar $A_i \rightarrow A_j\gamma$ por $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid ... \mid \delta_k\gamma$ (donde $A_j \rightarrow \delta_1 \mid ... \mid \delta_k \in P$)
    - Eliminar la recursión inmediata de A_i

Ejercicio 1:

a) $P = \{A \rightarrow A\{A\}A \mid \lambda\}$, $V_T = \{\ \{, \}\ \}$, $V_N = \{A\}$

Primero, notemos que no es posible hacer un parser recursivo predictivo porque hay recursión a izquierda en la producción $A \rightarrow A\{A\}A$. Vamos a hacer eliminación de la recursión inmediata (no hay no inmediata dado que sólo hay una variable, A).

$$A \rightarrow \lambda A'$$

$$A' \rightarrow \{A\}AA' \mid \lambda$$

Ahora ya no hay conflictos. Vamos a proceder a calcular los SDs para las distintas reglas y luego a armar los procedimientos para cada variable.

Tomemos la regla $A \rightarrow A'$ (es igual a $A \rightarrow \lambda A'$, antes se escribió el λ para explicitar todo el procedimiento). Como A' es anulable ($A' \overset{*}{\Rightarrow} \lambda$), obtenemos

$$SD(A \rightarrow A') = Primeros(A') \cup Siguientes(A) = \{\{\} \ \cup \ \{\$\}$$

Además,

$$SD(A' \rightarrow \{A\}AA') = \{\{\}$$

$$SD(A' \rightarrow \lambda) = Primeros(\lambda) \cup Siguientes(A') = \emptyset \cup Siguientes(A) = \{\$\}$$

```
Proc Main()
        if(ct != $) A();
        match('$');
        accept();
End

Proc A()
        A'()
End
```

```
Proc A'()
        if(ct == '{'){
        match('{');
        A();
        match('}');
        A();
        A'();
        }
        else{
                error();
        }
End
```

b) $P = \{A \to +AA \mid -AA \mid a\}$, $V_T = \{+, -, a\}$, $V_N = \{A\}$

Producción	SD
$A \to +AA$	$\{+\}$
$A \to -AA$	$\{-\}$
$A \to a$	$\{a\}$

```
Proc Main()
   A();
   match('$');
   accept();
End

Proc A()
   if(ct == '-'){
        match('-');
        A();
        A();
   }
   elseif(ct == '+'){
        match('+');
        A();
        A();
   }
   elseif(ct == 'a'){
    match('a');
   }
   else{
        error();
   }
End
```

c) $P = \{A \to 0A1 \mid 01\}$, $V_T = \{0, 1\}$, $V_N = \{A\}$

Hay que hacer *left-factorization*.

$$A \to 0A'$$
$$A' \to A1 \mid 1$$

Ahora ya está, no hay recursión no inmediata.

Ejercicio 2: Hecho en clase el 17/05.

Dada la siguiente gramática para la declaración de tipos en un lenguaje de programación: $G_1 = \langle \{S_0, T, S\}, \{\texttt{array, [,], of, .., int, char, num}, \uparrow\}, S_0, P\rangle$, con P:

$$S_0 \to T$$
$$T \to S \mid \ \uparrow S \mid \texttt{array of } T \mid \texttt{array } [S] \texttt{ of } T$$
$$S \to \texttt{int} \mid \texttt{char} \mid \texttt{num .. num}$$

(a) ¿Es posible crear un parser recursivo descendente predictivo para G_1? Si no, crear una gramática G_1' que genere el mismo lenguaje, y construir el parser para ella. Justificar.

No es posible hacer un parser descendente recursivo predictivo para la gramática, porque hay solapamiento entre los SDs de dos reglas del no terminal T.

Se lo resuelve aplicando factorización a la izquierda, para crear G_1'.

$G_1' = \langle \{S_0, T, T', S\}, \{\texttt{array, [,], of, .., int, char, num}, \uparrow\}, S_0, P\rangle$, con P:

$$
\begin{aligned}
S_0 &\to T \\
T &\to S \mid \ \uparrow S \mid \texttt{array}\ T' \\
T' &\to \texttt{of}\ T \mid [S]\ \texttt{of}\ T \\
S &\to \texttt{int} \mid \texttt{char} \mid \texttt{num .. num}
\end{aligned}
$$

Producción	SD
$S_0 \to T$	{int, char, num, ↑, array}
$T \to S$	{int, char, num}
$T \to \ \uparrow S$	{↑}
$T \to$ array T'	{array}
$T' \to$ of T	{of}
$T' \to [S]$ of T	{[}
$S \to$ int	{int}
$S \to$ char	{char}
$S \to$ num .. num	{num}

```
Proc Main()
   T();
   match('$');
   accept();
End

Proc T()
   if(ct in {int, char, num}){
        S();
   }
   elseif(ct == '^'){
        match('^');
        S();
   }
   elseif(ct == 'array'){
        match('array');
        T'();
   }
   else{
        error();
   }
End
```

```
Proc T'()
   if(ct == of){
        match('of');
        T();
   }
   elseif(ct == '['){
        match('[');
        S();
        match(']');
        match('of');
        T();
   }
   else{
        error();
   }
End

Proc S()
   if(ct == 'int'){
        match('int');
   }
   elseif(ct == 'char'){
        match('char');
   }
   elseif(ct == 'num'){
        match('num');
        match('..');
        match('num');
   }
   else{
        error();
   }
End
```

(b) Mostrar la derivación y el árbol correspondiente, paso a paso, para la siguiente cadena: `array [num..num] of ↑char`

$$S_0 \Rightarrow T \Rightarrow \texttt{array}\ T' \Rightarrow \texttt{array}\ [S]\ \texttt{of}\ T \Rightarrow \texttt{array}\ [\texttt{num}\ ..\ \texttt{num}]\ \texttt{of}\ T$$
$$\Rightarrow \texttt{array}\ [\texttt{num}\ ..\ \texttt{num}]\ \texttt{of}\ \uparrow S \Rightarrow \texttt{array}\ [\texttt{num}\ ..\ \texttt{num}]\ \texttt{of}\ \uparrow \texttt{char}$$

(c) ¿Qué ocurriría con la cadena anterior si se la procesara con un parser descendente que no fuera predictivo?

2. Parsers LL(1)

Una gramática $G =< V, T, P, S >$ es LL(1) sii

$$\forall\ (A \to \beta, A \to \gamma),\ \text{con}\ \beta \neq \gamma, SD(A \to \beta) \cap SD(A \to \gamma) = \emptyset$$

Ejercicio 3:

a) $G_1 =< \{S, A\}, \{a, b\}, P_1, S >$ con $P_1 = \{S \to aAS \mid b, A \to a \mid bSA\}$

Tengo que ver si se da que

$$SD(S \to aAS) \ \cap \ SD(S \to b) = \emptyset = SD(A \to a) \ \cap \ SD(A \to bSA)$$

$SD(S \to aAS) = Primeros(aAS) = \{a\}$ porque aAS no es anulable.

$SD(S \to b) = Primeros(b) = \{b\}$ porque b no es anulable.

La intersección de la izquierda es vacía.

$SD(A \to a) = \{a\}$ y $SD(A \to bSA) = \{b\}$ dan como resultado que la intersección de la derecha también sea vacía.

Concluimos que la gramática G_1 es LL(1).

b) $G_2 =< \{S\}, \{a, b\}, P_2, S >$ con $P_2 = \{S \to aaSbb \mid a \mid \lambda\}$

Afirmo que no es LL(1). Tomo el par de reglas $S \to aaSbb$ y $S \to a$.

$SD(S \to aaSbb) = Primeros(aaSbb) = \{a\}$ y $SD(S \to a) = Primeros(a) = \{a\}$ dan una intersección no vacía.

Hago *left-factoring* y obtengo

$$G_2' =< \{S, S'\}, \{a, b\}, P_2', S > \text{ con } P_2' = \{S \to aS' \mid \lambda, S' \to aSbb \mid \lambda\}$$

Tengo que verificar que

$$SD(S \to aS') \ \cap \ SD(S \to \lambda) = \emptyset = SD(S' \to aSbb) \ \cap \ SD(S' \to \lambda)$$

$$\begin{aligned} SD(S \to aS') \ \cap \ SD(S \to \lambda) &= Primeros(aS') \ \cap \ (Primeros(\lambda) \ \cup \ Siguientes(S)) \\ &= \{a\} \ \cap \ (\emptyset \ \cup \ \{\$\}) \\ &= \emptyset \end{aligned}$$

$$\begin{aligned} SD(S' \to aSbb) \ \cap \ SD(S' \to \lambda) &= Primeros(aSbb) \ \cap \ (Primeros(\lambda) \ \cup \ Siguientes(S')) \\ &= \{a\} \ \cap \ (\emptyset \ \cup \ Siguientes(S)) \\ &= \{a\} \ \cap \ (\emptyset \ \cup \ \{\$\}) \\ &= \emptyset \end{aligned}$$

Ejercicio 4: Crear el parser LL(1) para la gramática del Ejercicio 2.

Al ver los SD para las producciones nos damos cuenta de que la gramática del enunciado no es LL(1) (como vimos en el Ejercicio 2). La gramática corregida que propusimos, G_1', sí es LL(1) (como podemos ver en la tabla del Ejercicio 2).

Así, hacemos la tabla del parser LL(1).

NT	SÍMBOLO DE ENTRADA								
	`array`	`[`	`]`	`of`	`..`	`int`	`char`	`num`	$\uparrow$
S_0	$S_0 \to T$					$S_0 \to T$	$S_0 \to T$	$S_0 \to T$	$S_0 \to T$
T	`array` T'					$T \to S$	$T \to S$	$T \to S$	$T \to \ \uparrow S$
T'		$T' \to$ $[S]$ `of` T		$T' \to$ `of` T					
S						$S \to$ `int`	$S \to$ `char`	$S \to$ `num .. num`	

3. Gramáticas con conflictos

Ejercicio 5: Hecho en clase 18/05/2017.

Ejercicio 6: Dada la siguiente gramática que genera todas las expresiones regulares sobre los terminales {a, b, c}, indicar si es LL(1). Si no lo es, indicar qué cambios habría que realizarle para que lo fuera, sin modificar el lenguaje generado, o si se podría resolver de otra manera (e.g., ajustando la tabla del parser). $G_2 = \langle \{E\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{(}, \mathtt{)}, \mathtt{|}, *\}, E, P_2 \rangle$, con P_2:

$$E \to E\mathtt{|}E \mid EE \mid E^* \mid \mathtt{(}E\mathtt{)} \mid \mathtt{a} \mid \mathtt{b} \mid \mathtt{c}$$

Calculamos los SD para cada regla.

$$SD(E \to E|E) = Primeros(E|E) = \{\mathtt{(}, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$$

$$SD(E \to EE) = Primeros(EE) = \{\mathtt{(}, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$$

$$SD(E \to E^*) = Primeros(E^*) = \{\mathtt{(}, \mathtt{a}, \mathtt{b}, \mathtt{c}\}$$

$$SD(E \to \mathtt{(}E\mathtt{)}) = Primeros(\mathtt{(}E\mathtt{)}) = \{\mathtt{(}\}$$

$$SD(E \to \mathtt{a}) = Primeros(\mathtt{a}) = \{\mathtt{a}\}$$

$$SD(E \to \mathtt{b}) = Primeros(\mathtt{b}) = \{\mathtt{b}\}$$

$$SD(E \to \mathtt{c}) = Primeros(\mathtt{c}) = \{\mathtt{c}\}$$

Claramente no es LL(1).

NT	SÍMBOLO DE ENTRADA			
	(	a	b	c
E	E → E|E E → EE E → E^* E → (E)	E → E|E E → EE E → E^* E → a	E → E|E E → EE E → E^* E → b	E → E|E E → EE E → E^* E → c

No es posible resolver los conflictos ajustando la tabla del parser. Hay que modificar la gramática para marcar la precedencia de los operadores con distintos símbolos no terminales o eliminar la recursión inmediata.

Probamos eliminar la recursión inmediata.

$$E \to \mathtt{(}E\mathtt{)}E' \mid \mathtt{a}E' \mid \mathtt{b}E' \mid \mathtt{c}E'$$
$$E' \to \mathtt{|}EE' \mid EE' \mid {}^*E' \mid \lambda$$

Producción	SD
$E \to \mathtt{(}E\mathtt{)}E'$	{(}
$E \to \mathtt{a}E'$	{a}
$E \to \mathtt{b}E'S$	{b}
$E \to \mathtt{c}E'$	{c}
$E' \to \mathtt{|}EE'$	{| }
$E' \to EE'$	{(, a, b, c}
$E' \to {}^*E'$	{ * }
$E' \to \lambda$	{ |, (, a, b, c, * }

No funciona, vamos a probar agregando no terminales para marcar la precedencia.

$$
\begin{aligned}
E &\to E \,\texttt{|}\, E_1 \mid E_1 \\
E_1 &\to E_1 E_2 \mid E_2 \\
E_2 &\to {E_3}^* \mid E_3 \\
E_3 &\to (E) \mid \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{aligned}
$$

Producción	SD
$E \to E \,\texttt{|}\, E_1$	{(, a, b, c}
$E \to E_1$	{(, a, b, c}
$E_1 \to E_1 E_2$	{(, a, b, c}
$E_1 \to E_2$	{(, a, b, c}
${E_2 \to E_3}^*$	{(, a, b, c}
$E_2 \to E_3$	{(, a, b, c}
$E_3 \to \ (E)$	{(}
$E_3 \to \texttt{a}$	{a}
$E_3 \to \texttt{b}$	{b}
$E_3 \to \texttt{c}$	{c}

Eliminemos la recursión y hagamos factorización.

$$
\begin{aligned}
E &\to E_1 E' \\
E' &\to \texttt{|}\, E_1 E' \mid \lambda \\
E_1 &\to E_2 {E_1}' \\
{E_1}' &\to E_2 {E_1}' \mid \lambda \\
E_2 &\to E_3 {E_3}' \\
{E_3}' &\to \ ^* \mid \lambda \\
E_3 &\to (E) \mid \texttt{a} \mid \texttt{b} \mid \texttt{c}
\end{aligned}
$$

Producción	SD
$E \to E_1 E'$	$Primeros(E_1) = Primeros(E_2) = Primeros(E_3) = \{$`(, a, b, c`$\}$
$E' \to$ `|` $E_1 E'$	$\{$`|`$\}$
$E' \to \lambda$	$Siguientes(E') = Siguientes(E) = \{$`$`$\}$
$E_1 \to E_2 {E_1}'$	$Primeros(E_2) = \{$`(, a, b, c`$\}$
${E_1}' \to E_2 {E_1}'$	$Primeros(E_2) = \{$`(, a, b, c`$\}$
${E_1}' \to \lambda$	$Siguientes({E_1}') = Siguientes(E_1)$ $= Primeros(E') \cup Siguientes(E')$ $= \{$`|`$\} \cup Siguientes(E) = \{$`|, $`$\}$
$E_2 \to E_3 {E_3}'$	$Primeros(E_3) = \{$`(, a, b, c`$\}$
${E_3}' \to$ `*`	$\{$`*`$\}$
${E_3}' \to \lambda$	$Siguientes({E_3}') = Siguientes(E_2)$ $= Siguientes({E_1}') = Siguientes(E_1)$ $= Siguientes(E') = Siguientes(E) = \{$`$`$\}$
$E_3 \to$ `(`E`)`	$\{$`(`$\}$
$E_3 \to$ `a`	$\{$`a`$\}$
$E_3 \to$ `b`	$\{$`b`$\}$
$E_3 \to$ `c`	$\{$`c`$\}$

Esta gramática sí es LL(1), habría que ver que el lenguaje generado es el mismo que el de la gramática original.

4. Gramáticas extendidas (ELL) y parsers recursivos iterativos

Supongamos que tenemos la siguiente gramática:

$$L \to E \mid L, E$$
$$E \to n \mid (L)$$

Escribimos

$$L \to E(, E)^*$$
$$E \to n \mid (L)$$

y obtenemos los siguientes códigos:

$L \to E(, E)^*$

```
Proc L():
   E();
   while(tc == ','){
      match(',');
      E();
   }
End
```

$E \to n \mid (L)$

```
Proc E():
   if(tc == 'n'){match('n');}
   elseif(tc == '('){
      match('(');
      L();
      match(')');
   }
   else error();
End
```

4.1. Gramáticas extendidas

Las gramáticas extendidas y los métodos ELL permiten escribir usando ERs y generar parsers descendentes iterativos-recursivos.

Definición: Una gramática extendida es una tupla $\langle V_N, V_T, p, S\rangle$ con $p : V_N \to ER(V)$ ($V = V_N \cup V_T$). p puede ser una función porque se pueden combinar las producciones para un mismo no terminal con la unión.

Operadores: *, $^+$, ?. * se puede implementar con while, el $^+$ con do while y el ? con un if.

4.2. Generación de código y Transformación de gramáticas

Definimos por inducción la función $Cod(E)$ que recibe una expresión regular sobre V y devuelve el código correspondiente:

E	Cod(E)	E no extendida
λ	`skip;`	
`a`	`match('a')`	
$R?$	`if (tc in` Primeros(R) `) {` Cod(R) `}`	$A \to R \mid \lambda$
R^*	`while (tc in` Primeros(R) `) {` Cod(R) `}`	$A \to RA \mid \lambda$
R^+	`do {` Cod(R) `} while (tc in` Primeros(R) `)`	$A \to RR^*$
R_1R_2	Cod(R_1); Cod(R_2);	

$R_1 \mid R_2 \mid ... \mid R_n$	`if (tc in` SD($A \to R_1$) `) {` Cod(R_1) `}`
	`eif (tc in` SD($A \to R_2$) `) {` Cod(R_2) `}`
	...
	`else error`
		$A \to R_1 \mid R_2 \mid ...$

Notar que al agregar producciones de esta manera evitamos generar conflictos LL(1).

¿Cómo se usa?

El procedimiento consiste en reemplazar cada subexpresión (empezando por las más externas) por un no terminal nuevo y agregando las producciones que hagan falta.

Ejemplo: Si tenemos la producción $A \to a(bA?c)^*$ entonces se transforma en

$$A \to aA_1$$
$$A_1 \to bA?cA_1 \mid \lambda$$

y eso a su vez en

$$A \to aA_1$$
$$A_1 \to bA_2cA_1 \mid \lambda$$
$$A_2 \to A \mid \lambda$$

¿Es ELL(1)?

Si tenemos una Gramática Extendida EG y su GLC derivada es LL(1), decimos que EG es ELL(1) y el parser generado reconocerá exactamente las cadenas de L(EG).

Ejercicio 10: Dada la gramática $G_1 = \langle \{L, R, A, D\}, \{f, d, h, +, t\}, P, L \rangle$, con P:

$$\begin{aligned} L &\to R \mid R + L \\ R &\to A \mid RA \\ A &\to \texttt{d}L\texttt{h}D \mid \texttt{t} \mid \texttt{d}L\texttt{h} \\ D &\to \lambda \mid \texttt{f} \end{aligned}$$

(a) ¿Es G_1 ELL(1)? Justificar. Si no lo es, dar una gramática $G_1{}'$ que sea ELL(1) y tal que $L(G_1{}') = L(G_1)$ usando al menos una vez cada uno de los siguientes operadores: *, $^+$ y ?.
Describir cómo se llega a $G_1{}'$ a partir de G_1 y justificar por qué $G_1{}'$ es ELL(1).
La gramática G_1 es equivalente a

$$\begin{aligned} L &\to R(+R)^* \\ R &\to A^+ \\ A &\to \texttt{d}L\texttt{h}(D?) \mid \texttt{t} \\ D &\to \texttt{f}? \end{aligned}$$

que es equivalente a

$$L \to (\texttt{d}L\texttt{h}(f?) \mid \texttt{t})^+ \Big(+ (\texttt{d}L\texttt{h}(f?) \mid \texttt{t})^+ \Big)^*$$

$$\begin{aligned} L &\to A_1 A_2 \\ A_1 &\to A_3 A_4 \\ A_2 &\to +A_1 A_2 \mid \lambda \\ A_3 &\to \texttt{d}L\texttt{h}A_5 \mid \texttt{t} \\ A_4 &\to A_3 A_4 \mid \lambda \\ A_5 &\to \texttt{f} \mid \lambda \end{aligned}$$

Producción	SD
$L \to A_1 A_2$	$SD(A_1) = SD(A_3) = \{\texttt{d}, \texttt{t}\}$
$A_1 \to A_3 A_4$	$SD(A_3) = \{\texttt{d}, \texttt{t}\}$
$A_2 \to +A_1 A_2$	$\{+\}$
$A_2 \to \lambda$	$Siguientes(A_2) = Siguientes(L) = \{\$\}$
$A_3 \to \texttt{d}L\texttt{h}A_5$	$\{\texttt{d}\}$
$A_3 \to \texttt{t}$	$\{\texttt{t}\}$
$A_4 \to A_3 A_4$	$SD(A_3) = \{\texttt{d}, \texttt{t}\}$
$A_4 \to \lambda$	$Siguientes(A_4) = Siguientes(A_1)$ $= Primeros(A_2) \cup Siguientes(L)$ $= \{+\} \cup \{\$\} = \{+, \$\}$
$A_5 \to \texttt{f}$	$\{\texttt{f}\}$
$A_5 \to \lambda$	$Siguientes(A_5) = Siguientes(A_3)$ $= Primeros(A_4) \cup Siguientes(A_4)$ $= Primeros(A_3) \cup \{+, \$\}$ $= \{\texttt{d}, \texttt{t}\} \cup \{+, \$\} = \{\texttt{d}, \texttt{t}, +, \$\}$

(b) Dar el parser recursivo-iterativo de la gramática $G_1{}'$.

```
L():
    do{
        if(tc == d){
            match('d');
            L();
            match('h');
            if(tc == f) match('f');
        }
        elseif(tc == t){
            match('t');
        }
        else error();
    } while(tc in {d, t})
    while(tc == +){
        match('+');
        do{
            if(tc == d){
                match('d');
                L();
                match('h');
                if(tc == f) match('f');
            }
            elseif(tc == t){
                match('t');
            }
            else error();
        } while(tc in {d, t})
    }
```

Ejercicio 12: Dada la gramática extendida G_3, que genera un directorio teléfonico con entradas delimitadas por punto y coma (;), donde cada entrada tiene un nombre y un número de teléfono con, opcionalmente, un carácter y dos puntos (:) al comienzo, que indican bajo qué letra debe agendarse esa entrada (por defecto, cada entrada se agenda bajo la letra de comienzo del nombre, si lo hay). $G_3 = \langle \{A\}, \{\mathtt{a}, \mathtt{num}, \mathtt{,}, \mathtt{:}, \mathtt{;}\}, A, P_3 \rangle$, con P_3:

$$A \to \Big(\big((\mathtt{a} \ \mathtt{:} \ \mathtt{a}^*, \mathtt{num}) \mid (\mathtt{a}^*, \mathtt{num}) \big) \mathtt{;} \Big)^*$$

(a) Mostrar que G_3 no es ELL(1).

Si G_3 fuera ELL(1), la GLC derivada sería LL(1). Veamos que esto no es así.

$$\begin{aligned} A &\to A_1 \\ A_1 &\to (\mathtt{a} \ \mathtt{:} \ \mathtt{a}^*, \mathtt{num} \mid \mathtt{a}^*, \mathtt{num}) \mathtt{;} A_1 \mid \lambda \end{aligned}$$

$$\begin{aligned} A &\to A_1 \\ A_1 &\to (\mathtt{a} \ \mathtt{:} A_2, \mathtt{num} \mid \mathtt{a}^*, \mathtt{num}) \mathtt{;} A_1 \mid \lambda \\ A_2 &\to \mathtt{a} A_2 \mid \lambda \end{aligned}$$

$$
\begin{aligned}
A &\to A_1 \\
A_1 &\to (\texttt{a}\ :A_2, \texttt{num} \mid A_3, \texttt{num})\,;A_1 \mid \lambda \\
A_2 &\to \texttt{a}A_2 \mid \lambda \\
A_3 &\to \texttt{a}A_3 \mid \lambda
\end{aligned}
$$

$$
\begin{aligned}
A &\to A_1 \\
A_1 &\to A_4\,;A_1 \mid \lambda \\
A_2 &\to \texttt{a}A_2 \mid \lambda \\
A_3 &\to \texttt{a}A_3 \mid \lambda \\
A_4 &\to \texttt{a}\ :A_2, \texttt{num} \mid A_3, \texttt{num}
\end{aligned}
$$

Calculemos ahora los SD para las producciones de A_4.

$$SD(A_4 \to \texttt{a}\ :A_2, \texttt{num}) = \{\texttt{a}\}$$

$$SD(A_4 \to A_3, \texttt{num}) = Primeros(A_3) \cup Siguientes(A_3) = \{\texttt{a}\} \cup \{\texttt{,}\}$$

La intersección es $\{\texttt{a}\} \neq \emptyset$, por lo que concluimos que la GLC derivada no es LL(1).

(b) Crear una gramática de una sola producción para el lenguaje especificado, pero que sí sea ELL(1).

$$A \to \Big((\texttt{,}\ \texttt{num}) \mid \big(\texttt{a}\ (:\ \texttt{a}^*\texttt{,}\ \texttt{num} \mid \texttt{a}^*\texttt{,}\ \texttt{num})\big)\,;\Big)^*$$

5. Limitaciones de los parsers descendentes

6. Material

- Videos
  - Elimination of left recursion and left factoring the grammars: https://www.youtube.com/watch?v=3_VCoBfrt9c&t=4s
  - Introduction to parsers and LL(1) parsing: https://www.youtube.com/watch?v=N9UuAPU6DAg
  - Examples on how to find first and follow in LL(1): https://www.youtube.com/watch?v=_uSlP91jmTM&t=441s
  - Construction of LL(1) parsing table: https://www.youtube.com/watch?v=R1ZlWEZWMKk
  - Recursive descent parser: https://www.youtube.com/watch?v=SH5F-rwWEog
  - Operator grammar and Operator precedence parser: https://www.youtube.com/watch?v=n5UWAaw_byw&t=1s
- Libro del Dragón.
  - Top-Down Parsing: Capítulo 4.4.