

Teoría de Lenguajes

Parsers descendentes para gramáticas extendidas **ELL(1)**

Facultad de Ciencias Exactas y Naturales

Miércoles 24 de Mayo de 2017

Parsing Descendente

Hasta ahora vimos dos métodos de parsing descendente, que requieren que la gramática sea **LL(1)**:

Parsing Descendente

Hasta ahora vimos dos métodos de parsing descendente, que requieren que la gramática sea **LL(1)**:

- El predictivo recursivo

Parsing Descendente

Hasta ahora vimos dos métodos de parsing descendente, que requieren que la gramática sea **LL(1)**:

- El predictivo recursivo
- El basado en tablas

Parsing Descendente

Hasta ahora vimos dos métodos de parsing descendente, que requieren que la gramática sea **LL(1)**:

- El predictivo recursivo
- El basado en tablas

¿Y qué significa que sea **LL(1)**?

Parsing Descendente

Hasta ahora vimos dos métodos de parsing descendente, que requieren que la gramática sea **LL(1)**:

- El predictivo recursivo
- El basado en tablas

¿Y qué significa que sea **LL(1)**?

Que los símbolos directrices de las producciones de la gramática para un mismo no terminal sean disjuntos.

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas.

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L, n$$

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L , n$$

¿Es LL(1)?

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L, n$$

¿Es **LL(1)**? No, es recursiva a izquierda.

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L, n$$

¿Es **LL(1)**? No, es recursiva a izquierda. Eliminamos la recursión:

$$\begin{aligned} L &\rightarrow n L' \\ L' &\rightarrow, n L' \mid \lambda \end{aligned}$$

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L, n$$

¿Es **LL(1)**? No, es recursiva a izquierda. Eliminamos la recursión:

$$\begin{aligned} L &\rightarrow n L' \\ L' &\rightarrow, n L' \mid \lambda \end{aligned}$$

Ahora sí, pero si en lugar de eso devolvemos

$$L \rightarrow n \mid n, L$$

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L, n$$

¿Es **LL(1)**? No, es recursiva a izquierda. Eliminamos la recursión:

$$\begin{aligned} L &\rightarrow n L' \\ L' &\rightarrow, n L' \mid \lambda \end{aligned}$$

Ahora sí, pero si en lugar de eso devolvemos

$$L \rightarrow n \mid n, L$$

Tampoco es **LL(1)** porque dos producciones empiezan con lo mismo.

Ejemplo

Supongamos que queremos dar la gramática de una lista no vacía de números separados por comas. Una opción sería:

$$L \rightarrow n \mid L, n$$

¿Es **LL(1)**? No, es recursiva a izquierda. Eliminamos la recursión:

$$\begin{aligned} L &\rightarrow n L' \\ L' &\rightarrow, n L' \mid \lambda \end{aligned}$$

Ahora sí, pero si en lugar de eso devolvemos

$$L \rightarrow n \mid n, L$$

Tampoco es **LL(1)** porque dos producciones empiezan con lo mismo. Tendríamos que factorizar:

$$\begin{aligned} L &\rightarrow n L' \\ L' &\rightarrow, L \mid \lambda \end{aligned}$$

Ejemplo

Nos quedó una gramática complicada y difícil de leer para un lenguaje tan simple. Al ser un lenguaje regular, podríamos escribirlo como:

Ejemplo

Nos quedó una gramática complicada y difícil de leer para un lenguaje tan simple. Al ser un lenguaje regular, podríamos escribirlo como:

$$n(,n)^*$$

Ejemplo

Nos quedó una gramática complicada y difícil de leer para un lenguaje tan simple. Al ser un lenguaje regular, podríamos escribirlo como:

$$n(,n)^*$$

¿Cómo escribiríamos un programa para leer esto?

Ejemplo

Nos quedó una gramática complicada y difícil de leer para un lenguaje tan simple. Al ser un lenguaje regular, podríamos escribirlo como:

$$n(n)^*$$

¿Cómo escribiríamos un programa para leer esto?

```
match('n')
while(tc == ',')
    match(',')
    match('n')
end
```

Ejemplo

Nos quedó una gramática complicada y difícil de leer para un lenguaje tan simple. Al ser un lenguaje regular, podríamos escribirlo como:

$$n(,n)^*$$

¿Cómo escribiríamos un programa para leer esto?

```
match('n')  
while(tc == ',')  
    match(',')  
    match('n')  
end
```

¿Podremos usar algo así?

Otro caso

Ahora la lista acepta listas entre paréntesis

Otro caso

Ahora la lista acepta listas entre paréntesis

$$\begin{aligned}L &\rightarrow E \mid L, E \\ E &\rightarrow n \mid '(' L ')'\end{aligned}$$

Otro caso

Ahora la lista acepta listas entre paréntesis

$$\begin{aligned}L &\rightarrow E \mid L, E \\ E &\rightarrow n \mid '(' L ')'\end{aligned}$$

¿El lenguaje sigue siendo regular?

Otro caso

Ahora la lista acepta listas entre paréntesis

$$\begin{aligned}L &\rightarrow E \mid L, E \\ E &\rightarrow n \mid '(' L ')'\end{aligned}$$

¿El lenguaje sigue siendo regular? No

Otro caso

Ahora la lista acepta listas entre paréntesis

$$\begin{aligned}L &\rightarrow E \mid L , E \\ E &\rightarrow n \mid '(L)'\end{aligned}$$

¿El lenguaje sigue siendo regular? No

Estaría bueno poder escribir:

$$\begin{aligned}L &\rightarrow E (, E)^* \\ E &\rightarrow n \mid '(L)'\end{aligned}$$

Ejemplos

¿Cómo queda el código?

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

$$E \rightarrow n \mid '(' L ')'$$


Ejemplos

¿Cómo queda el código?

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

```
Proc L():  
  E()  
  while(tc == ',')  
    match(',')  
    E()  
  end  
end
```

$$E \rightarrow n \mid '(' L ')'$$

```
Proc E():  
  if(tc == 'n')  
    match('n')  
  else if(tc == '(')  
    match('(')  
    L()  
    match(')')  
  else  
    error   
  end  
end
```

Gramática extendida

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

Gramática extendida

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 $p: V_N \rightarrow \text{ER}(V)$

p puede ser una función porque se pueden combinar las producciones para un mismo no terminal con la unión.

Gramática extendida

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 $p: V_N \rightarrow \text{ER}(V)$

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`.

Generación de código (incompleto)

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

E	$\text{Cod}(E)$
λ	<code>skip;</code>
a	<code>match(a);</code>
$R?$	<code>if(tc in xxx){ Cod(R) }</code>
R^*	<code>while(tc in xxx){ Cod(R) }</code>
R^+	<code>do{ Cod(R) }while(tc in xxx)</code>
$R_1 R_2$	<code>Cod(R_1) ; Cod(R_2)</code>
$R_1 R_2 \dots R_n$	<code>if(tc in xxx)Cod(R_1) eif(tc in xxx)Cod(R_2) ... else error</code>

Generación de código (incompleto)

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

E	$\text{Cod}(E)$
λ	<code>skip;</code>
a	<code>match(a);</code>
$R?$	<code>if(tc in xxx){ Cod(R) }</code>
R^*	<code>while(tc in xxx){ Cod(R) }</code>
R^+	<code>do{ Cod(R) }while(tc in xxx)</code>
$R_1 R_2$	<code>Cod(R_1) ; Cod(R_2)</code>
$R_1 R_2 \dots R_n$	<code>if(tc in xxx)Cod(R_1) elseif(tc in xxx)Cod(R_2) ... else error</code>

¿Qué representa xxx en cada caso?

Transformación de gramáticas

Pasaje de gramática extendida a gramática común

E	$\text{Cod}(E)$	E no extendida
$R?$	<code>if(tc in xxx){ Cod(R) }</code>	$A \rightarrow R \lambda$
R^*	<code>while(tc in xxx){ Cod(R) }</code>	$A \rightarrow RA \lambda$
R^+	<code>do{ Cod(R) }while(tc in xxx)</code>	$A \rightarrow RR^*$
$R_1 R_2 \dots R_n$	<code>if(tc in xxx)Cod(R_1)</code> <code>eif(tc in xxx)Cod(R_2)</code> <code>...</code> <code>else error</code>	$A \rightarrow R_1 R_2 \dots$

Transformación de gramáticas

Cálculo de condiciones

E	$\text{Cod}(E)$	E no extendida
$R?$	<code>if(tc in Prim(R)){ Cod(R) }</code>	$A \rightarrow R \lambda$
R^*	<code>while(tc in Prim(R)){ Cod(R) }</code>	$A \rightarrow RA \lambda$
R^+	<code>do{ Cod(R) }while(tc in Prim(R))</code>	$A \rightarrow RR^*$
$R_1 R_2 \dots R_n$	<code>if(tc in SD($A \rightarrow R_1$))Cod(R_1)</code> <code>eif(tc in SD($A \rightarrow R_2$))Cod(R_2)</code> <code>...</code> <code>else error</code>	$A \rightarrow R_1 R_2 \dots$

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

Por ejemplo, si para producir β^* usáramos $B \rightarrow B\beta|\lambda$, estaríamos agregando una recursión a izquierda.

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

Por ejemplo, si para producir β^* usáramos $B \rightarrow B\beta|\lambda$, estaríamos agregando una recursión a izquierda.

¿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 1

Tenemos la producción:

$$A \rightarrow a(bA?c)^*$$

Ejemplo 1

Tenemos la producción:

$$A \rightarrow a(bA?c)^*$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA?cA_1|\lambda$$

Ejemplo 1

Tenemos la producción:

$$A \rightarrow a(bA?c)^*$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA?cA_1|\lambda$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA_2cA_1|\lambda$$

$$A_2 \rightarrow A|\lambda$$

Ejemplo 1

Tenemos la producción:

$$A \rightarrow a(bA?c)^*$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA?cA_1|\lambda$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA_2cA_1|\lambda$$

$$A_2 \rightarrow A|\lambda$$

¿Es **ELL**(1)?

Ejemplo 1

Tenemos la producción:

$$A \rightarrow a(bA?c)^*$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA?cA_1|\lambda$$

$$A \rightarrow aA_1$$

$$A_1 \rightarrow bA_2cA_1|\lambda$$

$$A_2 \rightarrow A|\lambda$$

¿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 $L(EG)$.

Ejemplo 2

Gramática común

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

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

$$F \rightarrow id \mid (E)$$

Ejemplo 2

Gramática común

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

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

$$F \rightarrow id \mid (E)$$

Gramática extendida

$$E \rightarrow T(+T)*$$

$$T \rightarrow F(*F)*$$

$$F \rightarrow id \mid (E)$$

Ejemplo 3

Resolvemos ejercicio 12 de la práctica 8

Fin

¿Preguntas?