Tecnológico de Monterrey

Desarrollo de aplicaciones avanzadas de Ciencias Computacionales

TC3002B.380

Actividad 3.2: Gramáticas libres de contexto

Presenta:
Alejandro Díaz Villagómez - A01276769

Docente:
Javier Abdul Córdoba Gándara

Fecha de entrega:
23 de mayo del 2024

Índice

Analizador LL

Un analizador LL es un tipo de analizador sintáctico utilizado en compiladores para analizar un texto según un conjunto de reglas de producción especificadas por una gramática formal para un lenguaje. El término LL se refiere a "Left-to-right, Leftmost derivation," que indica la forma en que el analizador procesa la entrada y construye la derivación. En pocas palabras, un analizador sintáctico LL es un autómata finito de pila que analiza la entrada desde la izquierda y construye el árbol sintáctico desde la raíz hasta las hojas.

Objetivos de un Analizador LL

- **Reconocimiento de la Estructura del Código**: El analizador LL toma una cadena de tokens (producida por el análisis léxico) y determina si esta cadena puede ser generada por la gramática del lenguaje de programación.
- **Construcción de Árboles de Derivación**: Durante el análisis, el parser puede construir un árbol de derivación o un árbol sintáctico, que representa la estructura sintáctica de la entrada.
- **Detección de Errores**: Si la entrada no es válida según la gramática, el parser debe detectar y reportar los errores sintácticos.

Componentes de un Analizador LL

- **Pila**: Se utiliza para mantener el estado del parser y los símbolos que se han analizado hasta el momento.
- **Tabla LL**: Contiene las reglas de producción que el parser debe aplicar basándose en el estado actual y el siguiente símbolo de entrada.

Tipos de Parsers LL

Existen varios tipos de parsers LL, diferenciados por la cantidad de lookahead (símbolos de anticipación) que utilizan y su capacidad para manejar diferentes gramáticas:

1. **LL(1)**: Utiliza un símbolo de lookahead y es el más común. Maneja gramáticas que pueden ser analizadas con una única mirada anticipada.
2. **LL(k)**: Utiliza k símbolos de lookahead. Puede manejar gramáticas más complejas que LL(1), pero es menos eficiente.
3. **Recursive Descent**: Es un tipo de parser LL(1) donde cada regla de producción se implementa como una función recursiva.

Ejercicio de analizador LL(1)

En aras de realizar este ejercicio, nos enfocaremos en la siguiente gramática:

S → I
S → o
I → i (E) S P
P → e S
P → ε
E → 0
E → 1

Primero, generaremos las tablas **FIRST** y **FOLLOW** de la gramática. Estas tablas son cruciales para determinar cómo construir la tabla de análisis LL(1). La tabla FIRST ayuda a identificar los posibles símbolos iniciales de las producciones, mientras que la tabla FOLLOW asegura que se consideran las posibles continuaciones en las derivaciones. Juntas, permiten rellenar correctamente la tabla LL(1), que guía al analizador sintáctico en la construcción de derivaciones válidas conforme a la gramática:

NT Symbols	First	Follow
S	{ o, i }	{ $, e, ε }
I	{ i }	{ $, e, ε }
P	{ e, ε }	{ $, e, ε }
E	{ 0, 1 }	{) }

Con apoyo de la tabla anterior, podemos crear la tabla de análisis LL(1):

NT	(	)	e	i	o	0	1	$
S				S → I	S → o			
I				I → i(E)SP				
P			P → eS P → ε					P → ε
E						E → 0	E → 1	

Como se puede apreciar, la gramática presenta una **ambigüedad** en tabla[P][e], lo que indica que no puede ser completamente analizada por un analizador LL(1). Sin embargo, para fines de este ejercicio, eliminaremos la regla P → ε:

NT	(	)	e	i	o	0	1	$
S				S → I	S → o			
I				I → i(E)SP				
P			P → eS					P → ε
E						E → 0	E → 1	

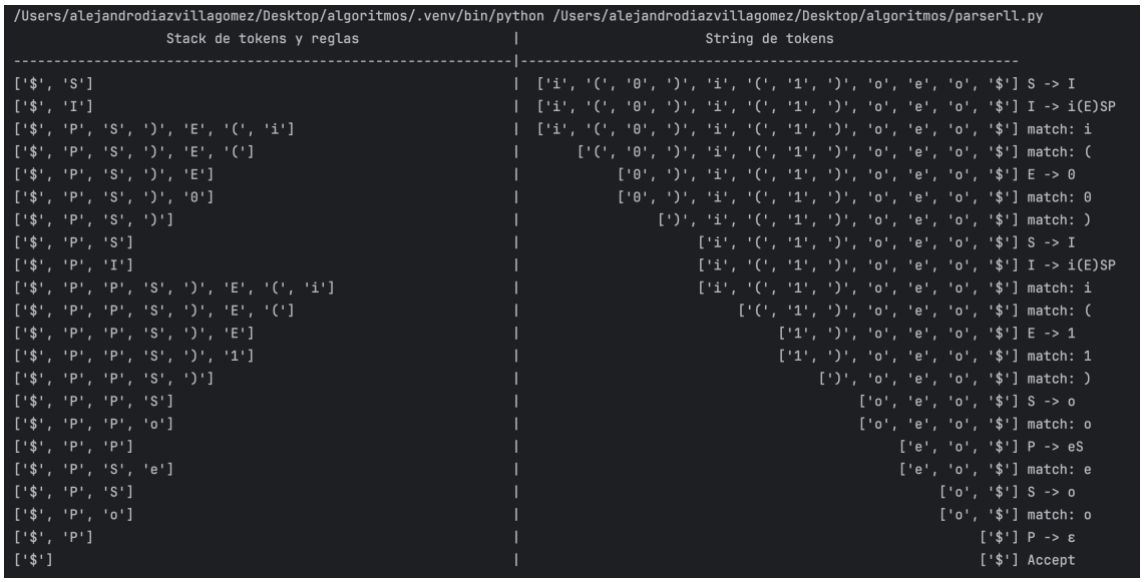
Para comprobar la gramática, vamos a realizar un análisis sintáctico manual y detallado de la cadena: **i(0)i(1)oeo**. El resultado es el siguiente:

Stack	Input	Action
S	i (0) i (1) o e o $	S → I
I	i (0) i (1) o e o $	I → i(E)SP
i(E)SP	i (0) i (1) o e o $	Match("i")
(E)SP	(0) i (1) o e o $	Match("(")
E)SP	0) i (1) o e o $	E → 0
0)SP	0) i (1) o e o $	Match("0")
)SP	) i (1) o e o $	Match(")")
SP	i (1) o e o $	S → I
IP	i (1) o e o $	I → i(E)SP
i(E)SPP	i (1) o e o $	Match("i")
(E)SPP	(1) o e o $	Match("(")
E)SPP	1) o e o $	E → 1
1)SPP	1) o e o $	Match("1")
)SPP	) o e o $	Match(")")
SPP	o e o $	S → o
oPP	o e o $	Match("o")
PP	e o $	P → eS
eSP	e o $	Match("e")
SP	o $	S → o
oP	o $	Match("o")
P	$	P → ε
ε	$	Accept

Como se puede ver en el análisis detallado, la cadena **i(0)i(1)oeo** se analiza correctamente. Durante el proceso, la pila se manipula correctamente, lo que permite reducirla hasta alcanzar un estado de aceptación.

Este resultado confirma que la gramática propuesta puede aceptar la cadena dada, validando tanto la corrección de la gramática como la eficacia del analizador y la tabla LL(1) generada. Esto demuestra que el análisis sintáctico se ha realizado con éxito, asegurando que la estructura de la cadena se ajusta a las reglas definidas por la gramática.

Al correr el programa en Python, obtenemos el mismo resultado satisfactorio, lo que reafirma la capacidad de la gramática y del analizador para procesar y validar la cadena correctamente:



```
/Users/alejandrodiazvillagomez/Desktop/algoritmos/.venv/bin/python /Users/alejandrodiazvillagomez/Desktop/algoritmos/parserll.py
                  Stack de tokens y reglas                  |                      String de tokens
------------------------------------------------------------|-------------------------------------------------------------
['$', 'S']                                                  |  ['i', '(', '0', ')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] S -> I
['$', 'I']                                                  |  ['i', '(', '0', ')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] I -> i(E)SP
['$', 'P', 'S', ')', 'E', '(', 'i']                         |  ['i', '(', '0', ')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] match: i
['$', 'P', 'S', ')', 'E', '(']                              |      ['(', '0', ')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] match: (
['$', 'P', 'S', ')', 'E']                                   |           ['0', ')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] E -> 0
['$', 'P', 'S', ')', '0']                                   |           ['0', ')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] match: 0
['$', 'P', 'S', ')']                                        |               [')', 'i', '(', '1', ')', 'o', 'e', 'o', '$'] match: )
['$', 'P', 'S']                                             |                   ['i', '(', '1', ')', 'o', 'e', 'o', '$'] S -> I
['$', 'P', 'I']                                             |                   ['i', '(', '1', ')', 'o', 'e', 'o', '$'] I -> i(E)SP
['$', 'P', 'P', 'S', ')', 'E', '(', 'i']                    |                   ['i', '(', '1', ')', 'o', 'e', 'o', '$'] match: i
['$', 'P', 'P', 'S', ')', 'E', '(']                         |                       ['(', '1', ')', 'o', 'e', 'o', '$'] match: (
['$', 'P', 'P', 'S', ')', 'E']                              |                           ['1', ')', 'o', 'e', 'o', '$'] E -> 1
['$', 'P', 'P', 'S', ')', '1']                              |                           ['1', ')', 'o', 'e', 'o', '$'] match: 1
['$', 'P', 'P', 'S', ')']                                   |                               [')', 'o', 'e', 'o', '$'] match: )
['$', 'P', 'P', 'S']                                        |                                   ['o', 'e', 'o', '$'] S -> o
['$', 'P', 'P', 'o']                                        |                                   ['o', 'e', 'o', '$'] match: o
['$', 'P', 'P']                                             |                                       ['e', 'o', '$'] P -> eS
['$', 'P', 'S', 'e']                                        |                                       ['e', 'o', '$'] match: e
['$', 'P', 'S']                                             |                                           ['o', '$'] S -> o
['$', 'P', 'o']                                             |                                           ['o', '$'] match: o
['$', 'P']                                                  |                                               ['$'] P -> ε
['$']                                                       |                                               ['$'] Accept
```

Analizador LR

Un LR parser (analizador LR) es un tipo de analizador sintáctico utilizado en compiladores para analizar un texto de acuerdo a conjunto de reglas de producción especificadas por una gramática formal para un lenguaje. El término LR se refiere a "Left-to-right, Rightmost derivation in reverse," que indica la forma en que el analizador procesa la entrada y construye la derivación. Básicamente, un analizador sintáctico LR es un autómata finito de pila que reconoce prefijos viables y va construyendo el árbol sintáctico desde las hojas hasta la raíz.

Objetivos de un analizador LR

- **Reconocimiento de la Estructura del Código**: El LR parser toma una cadena de tokens (producida por el análisis léxico) y determina si esta cadena puede ser generada por la gramática del lenguaje de programación.

- **Construcción de Árboles de Derivación**: Durante el análisis, el parser puede construir un árbol de derivación o un árbol sintáctico, que representa la estructura sintáctica de la entrada.

- **Detección de Errores**: Si la entrada no es válida según la gramática, el parser debe detectar y reportar los errores sintácticos.

Componentes de un analizador LR

- **Pila**: Se utiliza para mantener el estado del parser y los símbolos que se han analizado hasta el momento.

- **Tabla LR**: Contiene dos partes:
  - **Tabla de acción**: Define las acciones que el parser debe realizar (desplazamiento/shift, reducción/reduce, aceptación/accept o error) basándose en el estado actual y el siguiente símbolo de entrada.

  - **Tabla de ir_a (goto)**: Define las transiciones entre estados basándose en los no terminales.

Tipos de Parsers LR

Existen varios tipos de parsers LR, diferenciados por la cantidad de lookahead (símbolos de anticipación) que utilizan y su complejidad:

1. **LR(0)**: No utiliza símbolos de lookahead. Es el más simple pero puede manejar menos gramáticas.

2. **SLR(1) (Simple LR)**: Utiliza un símbolo de lookahead y es una extensión del LR(0).

3. **LALR(1) (Look-Ahead LR)**: Utiliza un símbolo de lookahead y combina estados que tienen las mismas producciones sin lookahead. Es más eficiente en términos de memoria.

4. **LR(1)**: Utiliza un símbolo de lookahead y maneja un conjunto más amplio de gramáticas que SLR(1) y LALR(1), pero es más complejo y consume más memoria.

Ejercicio de analizador LR(0)

Considerando la siguiente gramática, vamos a generar su autómata y su tabla LR:

E → (L)
E → a
L → L,E
L → E

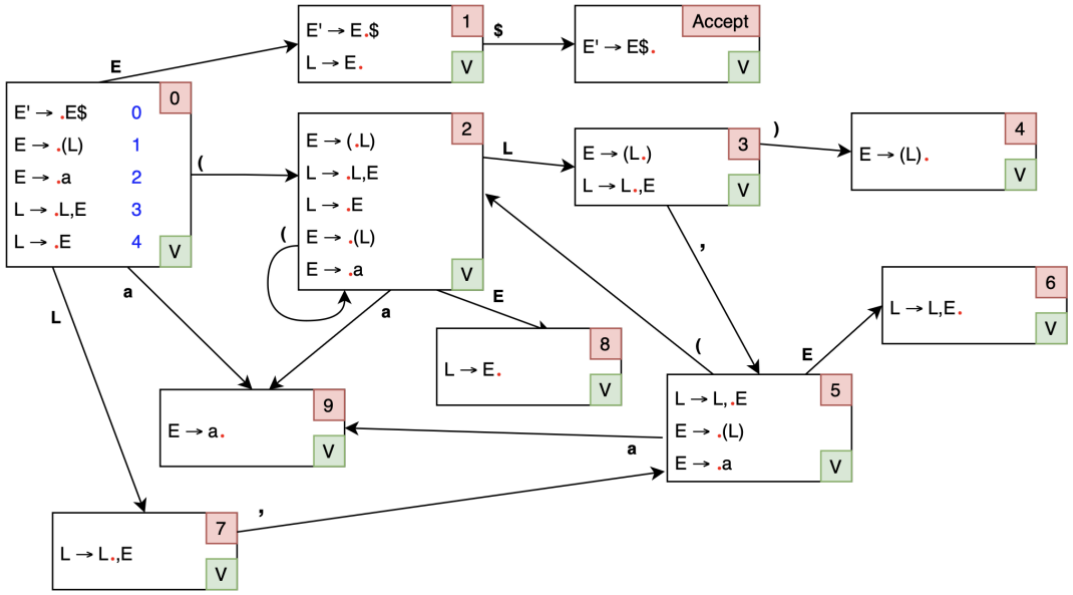
Recordemos que una gramática LR(0) bien formada siempre debe tener un estado de aceptación en su autómata. Este estado es esencial para el correcto funcionamiento del analizador sintáctico, permitiendo reconocer cuándo una cadena de entrada ha sido completamente y correctamente analizada según las reglas de la gramática. Si una gramática LR(0) no tiene un estado de aceptación, esto indica un problema en la construcción del autómata o en la definición de la gramática misma. Por ello, debemos **aumentar la gramática** añadiendo una nueva regla inicial: E' → E$.

Nos apoyaremos del concepto de **item** para la creación del autómata y de la tabla que representa esta gramática. Los items son fundamentales para construir autómatas LR(0) y representan los estados iniciales, intermedios y finales del proceso de análisis sintáctico. Por ejemplo, un item para la producción $E \rightarrow (L)$ es:

- $E \rightarrow .(L)$
- $E \rightarrow (.L)$
- $E \rightarrow (L.)$
- $E \rightarrow (L.)$
- $E \rightarrow (L).$

Cada item muestra una posible posición del análisis, ayudando a construir el autómata LR(0) que define cómo se procesa la cadena de entrada.

El autómata resultante de la gramática en cuestión es:



Con apoyo del autómata, podemos generar la tabla LR que representa la gramática propuesta:

	Action					GOTO	
State	(	)	a	,	$	E	L
0	s2		s9			1	7
1					acc		
2	s2		s9			8	3
3		s4		s5			
4	r1	r1	r1	r1	r1		
5	s2		s9			6	
6	r3	r3	r3	r3	r3		
7				s5			
8	r4	r4	r4	r4	r4		
9	r2	r2	r2	r2	r2		

Finalmente, podemos llevar a cabo de manera detallada (paso a paso) de como la cadena **((a),a,(a,a))** se parsea de acuerdo al automáta y la tabla obtenidos:

Stack	Input	Action
0	((a),a,(a,a))$	Shift(2)
02(	(a),a,(a,a))$	Shift(2)
02(2(	a),a,(a,a))$	Shift(9)
02(2(9a	),a,(a,a))$	Reduce(2) - E → a
02(2(8E	),a,(a,a))$	Reduce(4) - L → E
02(2(3L	),a,(a,a))$	Shift(4)
02(2(3L4)	,a,(a,a))$	Reduce(1) - E → (L)
02(8E	,a,(a,a))$	Reduce(4) - L → E
02(3L	,a,(a,a))$	Shift(5)
02(3L5,	a,(a,a))$	Shift(9)
02(3L5,9a	,(a,a))$	Reduce(2) - E → a
02(3L5,6E	,(a,a))$	Reduce(3) - L → L, E
02(3L	,(a,a))$	Shift(5)
02(3L5,	(a,a))$	Shift(2)
02(3L5,2(	a,a))$	Shift(9)

$02(3^{L}5,2(9^{a}$	,a))$	Reduce(2) - E → a
$02(3^{L}5,2(8^{E}$	,a))$	Reduce(4) - L → E
$02(3^{L}5,2(3^{L}$	,a))$	Shift(5)
$02(3^{L}5,2(3^{L}5,$	a))$	Shift(9)
$02(3^{L}5,2(3^{L}5,9^{a}$	))$	Reduce(2) - E → a
$02(3^{L}5,2(3^{L}5,6^{E}$	))$	Reduce(3) - L → L, E
$02(3^{L}5,2(3^{L}$	))$	Shift(4)
$02(3^{L}5,2(3^{L}4)$	)$	Reduce(1) - E → (L)
$02(3^{L}5,6^{E}$	)$	Reduce(3) - L → L, E
$02(3^{L}$	)$	Shift(4)
$02(3^{L}4)$	$	Reduce(1) - E → (L)
$0^{E}1$	$	Accept

Como se puede observar en el análisis detallado, la cadena **((*a*),*a*,(*a*,*a*))((*a*),*a*,(*a*,*a*))** se parsea correctamente. A lo largo del proceso, la pila se manipula siguiendo las acciones especificadas en la tabla LR, realizando desplazamientos y reducciones en los momentos apropiados. Finalmente, la pila se reduce hasta alcanzar un estado de aceptación (accept).

Este resultado confirma que la gramática propuesta es capaz de aceptar la cadena dada, validando tanto la corrección de la gramática como la eficacia del autómata y la tabla LR generada. Esto demuestra que el análisis sintáctico se ha llevado a cabo con éxito, asegurando que la estructura de la cadena se ajusta a las reglas definidas por la gramática.

Al correr el programa en Python, obtenemos el mismo resultado:

Links relevantes

Para más detalles sobre los gráficos y tablas obtenidos, así como el código completo de la solución,se encuentran los siguientes enlaces:

- *Diagramas-Tablas.drawio*
- *Repo-GitHub.com*

Referencias

- Barenbaum, P. (2020, octubre 8). PGC04 - 2 Análisis sintáctico LL(1) [Video]. YouTube. https://www.youtube.com/watch?v=WxILX_8GdIg
- Neso Academy. (2023, marzo 26). LL(1) Parsing [Video]. YouTube. https://www.youtube.com/watch?v=clkHOgZUGWU
- Neso Academy. (2023, marzo 24). LL(1) Parsing Table [Video]. YouTube. https://www.youtube.com/watch?v=DT-cbznw9aY
- Anita R. (2020, febrero 17). Compiler Design: Predictive Parsing-LL(1) [Video]. YouTube. https://www.youtube.com/watch?v=QoOALbef3ZM
- Lajpop, K. (2020, abril 21). Análisis sintáctico: LL1 parser [Video]. YouTube. https://www.youtube.com/watch?v=tHm8rnYuvLo
- Barenbaum, P. (2020, octubre 8). PGC05 - 2 Análisis sintáctico LR(0) [Video]. YouTube. https://www.youtube.com/watch?v=BHZ-AFaJPvA
- Ravula, R. (2014, mayo 23). CD | Parsers | LR parsing, LR(0) items and LR(0) parsing table | Ravindrababu Ravula | Free GATE CS [Video]. YouTube. https://www.youtube.com/watch?v=APJ_Eh60Qwo
- CSE concepts with Parinita. (2019, octubre 10). 15. LR(0) parsing in Compiler Design | LR 0 Parser Example | Canonical collection of lr(0) items table [Video]. YouTube. https://www.youtube.com/watch?v=zaomF0mznQo