

Reporte: Práctica 6

Barrera Pérez Carlos Tonatihu
Profesor: Saucedo Delgado Rafael Norman
Compiladores
Grupo: 3CM6

12 de diciembre de 2017

Índice

1. Introducción	2
2. Desarrollo	4
3. Resultados	10
3.1. Prueba 1	10
3.2. Prueba 2	11
3.3. Prueba 3	11
4. Conclusiones	13

1. Introducción

Para la construcción de la tabla LL(1) es necesario el uso de dos métodos los cuales son *PRIMERO* y *SIGUIENTE*, estos dos métodos nos permiten elegir que producción se utilizara con base al símbolo de entrada. [1]

Estas dos operaciones se definen de la siguiente forma.

- *PRIMERO*(α). Donde α es una cadena de símbolos gramaticales es el conjunto de terminales que empiezan la las cadenas derivadas a partir de α .
- *SIGUIENTE*(A). Donde A es un no terminal es el conjunto de terminales a que pueden aparecer de inmediato a la derecha de A .

Ahora bien, para calcular *PRIMERO*(X) donde X es un símbolo gramatical se aplican las siguientes reglas hasta que no puedan agregarse mas terminales o ϵ .

- Si X es un terminal entonces *PRIMERO*(X) = $\{X\}$.
- Si X es un no terminal entonces por cada producción de X

$$X \rightarrow Y_1 Y_2 Y_3 \dots Y_m$$

- Agregar *PRIMERO*(Y_i) a *PRIMERO*(X).
- Si *PRIMERO*(Y_i) contiene ϵ avanza i .

Por otro lado para calcular *SIGUIENTE*(N) se siguen las siguientes reglas.

- Si N es inicial. Agregar \$ a *SIGUIENTE*(N).
- Si $A \rightarrow \alpha N$. Agregar *SIGUIENTE*(A).
- Si $A \rightarrow \alpha N \beta$. Agregar *PRIMERO*(β).
- Si $A \rightarrow \alpha N$ y $\epsilon \in \text{PRIMERO}(\beta)$. Agregar *SIGUIENTE*(A).

No TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Figura 1: Tabla de análisis sintáctico M [1].

Ya que se tiene la definición de las funciones *PRIMERO* y *SIGUIENTE* se tiene que construir la tabla M como la mostrada en la figura 1 para poder realizar el análisis sintáctico, esta creación se realiza utilizando el siguiente algoritmo. [1]

1. Por cada producción $A \rightarrow \alpha$
 - a) Por cada a en primero $PRIMERO(\alpha)$.
 - 1) Agregar $A \rightarrow \alpha$ en $M[A, a]$.
 - b) Si $\epsilon \in PRIMERO(\alpha)$
 - 1) Agregar $A \rightarrow \alpha$ en $M[A, c]$ para cada c en $SIGUIENTE(A)$.

Un ejemplo de la construcción de esta tabla se encuentra en la figura 1 dicha tabla se construyo utilizando la siguiente gramática.

$$\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}$$

2. Desarrollo

Para la creación de la tabla LL(1) se definieron tres clases para poder hacer uso de la herencia y con ello poder separar el código de una forma que sea más fácil de entender además de que permita la reutilización de este.

Esta es una clase muy importante ya que facilita la manipulación de gramáticas ya que permite un acceso rápido a los componentes de una gramática libre de contexto.

Archivo: gramatica.py

```
1  import re
2
3
4  class Gramatica:
5      """Clase que almacena los componentes de una gramatica:
6          - terminales
7          - no terminales
8          - producciones
9          - simbolo inicial
10         Tambien se encarga de leer la gramatica del archivo
11         y almacenarla"""
12     def __init__(self, archivo):
13         self.nombre_archivo = archivo
14         self.no_terminales = dict()
15         self.terminales = dict()
16         self.inicial = None
17         self.gramatica = dict()
18
19     def leer_archivo(self):
20         """Metodo que lee el archivo y obtiene
21         los componentes de la gramatica"""
22         archivo = open(self.nombre_archivo, 'r')
23         primera = 0
24         j = 0
25         for linea in archivo:
26             if primera == 0:
27                 self.obtener_no_terminales(linea)
28                 primera = 1
29                 continue
30             if primera == 1:
31                 self.inicial = linea[0]
32                 primera = 2
33             j = self.obtener_produccion(linea, j)
34             self.terminales.update({"$": j})
35
36     def obtener_produccion(self, linea, j):
37         """Metodo cada produccion"""
38         izq = linea[0]
```

```

39     der = linea[3:]
40     der = re.match("[^\n]*", der).group()
41     if izq not in self.gramatica:
42         self.gramatica.update({izq: {
43             "producciones": list(),
44             "primero": False,
45             "siguiente": False
46         }})
47     self.gramatica.get(izq).get("producciones").append(der)
48     for c in der:
49         busqueda = re.match("[a-df-z\\(\\)\\+\\-\\*]", c) is not None
50         if busqueda and c not in self.terminals:
51             self.terminals.update({c: j})
52             j += 1
53     return j
54
55     def obtener_no_terminals(self, linea):
56         """Obtiene los terminales de la primera linea del archivo"""
57         i = 0
58         for c in linea:
59             busqueda = re.match("[A-Z]", c) is not None
60             if busqueda and c not in self.no_terminals:
61                 self.no_terminals.update({c: i})
62                 i += 1

```

Esta clase es donde están declaradas las funciones *PRIMERO* y *SIGUIENTE*. Hereda de clase *Gramatica* para poder trabajar con alguna gramática y facilitar su manipulación.

Archivo: auxiliares.py

```

1  from gramatica import Gramatica
2  import re
3
4
5  class Auxiliares(Gramatica):
6      """Clase que contiene la implementacion de los
7      metodos primero y siguiente"""
8      def __init__(self, archivo):
9          """Se envia el nombre del archivo
10          a la clase padre"""
11          super(Auxiliares, self).__init__(archivo)
12
13      def es_epsilon(self, A):
14          return A == 'ε'
15
16      def es_terminal(self, A):
17          return A not in self.gramatica
18
19      def es_inicial(self, S):

```

```

20         return self.inicial == S
21
22     def primero(self, A):
23         """Metodo que calcula primero de una cadena"""
24         conjunto = set()
25         for a in A:
26             if a in self.gramatica:
27                 self.gramatica.get(a)["primero"] = False
28                 conjunto_extra = self.P(a)
29                 conjunto.update(conjunto_extra)
30                 if 'e' not in conjunto_extra:
31                     if 'e' in conjunto:
32                         conjunto.remove('e')
33                     break
34         return conjunto
35
36     def P(self, A):
37         """Metodo que calcula primero de un solo simbolo"""
38         conjunto = set()
39         if self.es_terminal(A) or self.es_epsilon(A):
40             conjunto.add(A)
41         else:
42             if self.gramatica.get(A).get("primero"):
43                 return conjunto
44             else:
45                 self.gramatica.get(A)["primero"] = True
46
47             producciones = self.gramatica.get(A).get("producciones")
48             for produccion in producciones:
49                 i = 0
50                 while i < len(produccion):
51                     extra = self.P(produccion[i])
52                     conjunto.update(extra)
53                     if 'e' in extra:
54                         i += 1
55                     else:
56                         break
57         return conjunto
58
59     def siguiente(self, N):
60         """Metodo para el calculo de siguiente
61         se inicializan los banderas que indican si ya
62         se calculo siguiente"""
63         for clave, valor in self.gramatica.items():
64             valor["siguiente"] = False
65         return self.S(N)
66
67     def S(self, N):

```

```

68     """Calculo de siguiente"""
69     conjunto = set()
70     if not self.es_terminal(N) and self.gramatica.get(N).get("siguiente"):
71         return conjunto
72     else:
73         if not self.es_terminal(N):
74             self.gramatica.get(N)["siguiente"] = True
75
76     if self.es_inicial(N):
77         conjunto.add('$')
78     # A -> xN
79     no_terminales = self.obtener_izquierda(N)
80     if len(no_terminales) != 0:
81         for n in no_terminales:
82             conjunto.update(self.S(n))
83     # A -> xNy
84     no_terminales = self.obtener_derecha(N)
85     if len(no_terminales) != 0:
86         for simbolo in no_terminales:
87             primero = self.primeros(simbolo.get("cadena"))
88             if 'e' in primero:
89                 primero.remove('e')
90                 conjunto.update(self.S(simbolo.get("clave")))
91             conjunto.update(primero)
92     if not self.es_terminal(N):
93         self.gramatica.get(N)["siguiente"] = False
94     return conjunto
95
96 def obtener_izquierda(self, N):
97     """Obtiene la parte izquierda de una produccion"""
98     claves = list()
99     for clave, valor in self.gramatica.items():
100         for v in valor.get("producciones"):
101             if N == v[len(v)-1]:
102                 claves.append(clave)
103     return claves
104
105 def obtener_derecha(self, N):
106     """Obtiene la parte derecha de una produccion"""
107     simbolos = list()
108     for clave, valor in self.gramatica.items():
109         for v in valor.get("producciones"):
110             for m in re.finditer(N, v):
111                 if m.start() != len(v)-1:
112                     simbolos.append({
113                         "clave": clave,
114                         "cadena": v[m.start()+1:]
115                     })

```


Esta clase es la encargada de construir la tabla con el algoritmo antes mencionado, hereda de la clase Auxiliares para poder utilizar las funciones de *PRIMERO* y *SIGUIENTE*.

Archivo: tabla.py

```

1 from auxiliares import Auxiliares
2
3
4 class TablaLL(Auxiliares):
5     """Clase para la creacion y despliegue de la tabla LL(1)
6     utilizando los metodos contenidos en la clase padre Auxiliares"""
7     def __init__(self, archivo):
8         """Se recibe el nombre del archivo y se pasa a la clase padres"""
9         super(TablaLL, self).__init__(archivo)
10        self.leer_archivo()
11        self.num_filas = len(self.no_terminales)
12        self.num_colum = len(self.terminales)
13        self.tabla = [[None] * self.num_colum for i in range(self.num_filas)]
14
15    def construir_tabla(self):
16        """Implementacion del algoritmo para el
17        llenado de la tabla"""
18        produccion_num = 1
19        for clave, valor in self.gramatica.items():
20            for produccion in valor.get("producciones"):
21                primeros = self.primeros(produccion)
22                for a in primeros:
23                    if a != "ε":
24                        self.agregar_elemento(clave, a, produccion_num)
25                    if "ε" in primeros:
26                        siguientes = self.siguiete(clave)
27                        for c in siguientes:
28                            self.agregar_elemento(clave, c, produccion_num)
29
30                produccion_num += 1
31
32    def agregar_elemento(self, A, a, num):
33        """Metodo que agrega un elemento a la tabla"""
34        i = self.no_terminales.get(A)
35        j = self.terminales.get(a)
36        if self.tabla[i][j] is None:
37            self.tabla[i][j] = set()
38        self.tabla[i][j].add(num)
39
40    def mostrar_tabla(self):
41        """Metodo que muestra la tabla"""
42        print("Tabla LL(1):")

```

```
43     print("    ", end="")
44     for t in self.terminales.keys():
45         print(t, end="\t")
46     print("")
47     for f, n in zip(self.tabla, self.no_terminales.keys()):
48         for c in f:
49             print(c, end="\t")
50         print(n)
```

Este archivo es en el cual se realizan las pruebas, después de ejecutarlo se introduce el nombre del archivo que contiene la gramática.

Archivo: prueba.py

```
1 from tabla import TablaLL
2 # creacion de la tabla LL(1)
3 gramatica = input("Nombre del archivo: ")
4 tabla = TablaLL(gramatica)
5 tabla.construir_tabla()
6 tabla.mostrar_tabla()
```

3. Resultados

Para comprobar que la creación de la tabla se realiza de forma correcta se realizaron 3 pruebas con gramáticas diferentes. Dichas gramáticas se ingresan mediante un archivo de texto que contiene dicha una gramática, además de tener la gramática el archivo tiene los símbolos no terminales en la primera línea del archivo.

3.1. Prueba 1

Gramática usada en esta prueba.

$$S \rightarrow abBCa \quad (1)$$

$$B \rightarrow bACA \quad (2)$$

$$B \rightarrow aC \quad (3)$$

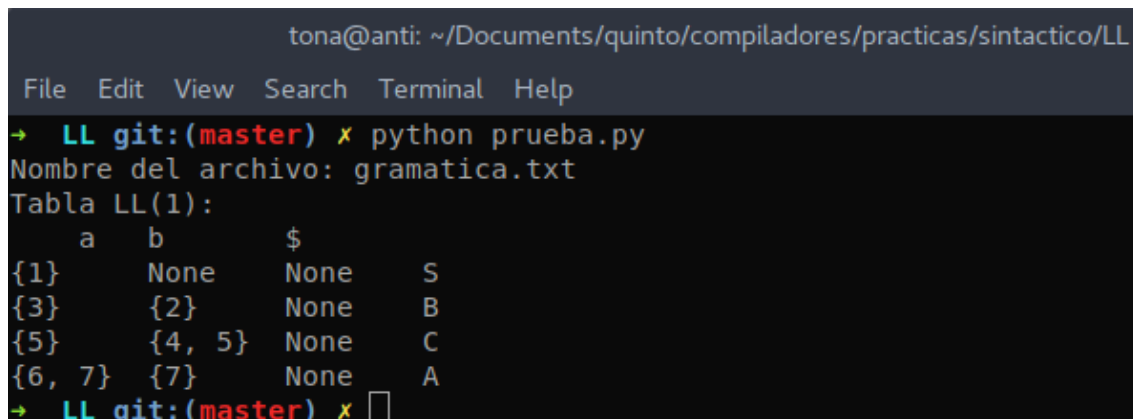
$$C \rightarrow bAbS \quad (4)$$

$$C \rightarrow e \quad (5)$$

$$A \rightarrow a \quad (6)$$

$$A \rightarrow e \quad (7)$$

Salida del programa.



```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LL
File Edit View Search Terminal Help
→ LL git:(master) x python prueba.py
Nombre del archivo: gramatica.txt
Tabla LL(1):
  a    b    $
{1}   None None  S
{3}   {2} None  B
{5}   {4, 5} None C
{6, 7} {7} None  A
→ LL git:(master) x
```

Figura 2: Tabla LL(1).

Después de observar esta tabla podemos concluir que esta gramática produciría problemas debido a que hay más de un elemento en una celda de la tabla.

3.2. Prueba 2

Gramática usada en esta prueba.

$$S \rightarrow iEtSD \quad (1)$$

$$S \rightarrow a \quad (2)$$

$$D \rightarrow oS \quad (3)$$

$$D \rightarrow e \quad (4)$$

$$E \rightarrow b \quad (5)$$

Salida del programa.

```
tona@anti: ~/Documents/quinto/compiladores/practicass/sintactico/LL
File Edit View Search Terminal Help
→ LL git:(master) x python prueba.py
Nombre del archivo: gramatica2.txt
Tabla LL(1):
      i      t      a      o      b      $      S
{1}    None   {2}    None   None   None    S
None   None   None   {3, 4} None   {4}    D
None   None   None   None   {5}   None    E
→ LL git:(master) x
```

Figura 3: Tabla LL(1).

Después de observar esta tabla podemos concluir que esta gramática produciría problemas debido a que hay más de un elemento en una celda de la tabla.

3.3. Prueba 3

Gramática usada en esta prueba.

$$E \rightarrow TR \quad (1)$$

$$R \rightarrow +TR \quad (2)$$

$$R \rightarrow e \quad (3)$$

$$T \rightarrow FY \quad (4)$$

$$Y \rightarrow *FY \quad (5)$$

$$Y \rightarrow e \quad (6)$$

$$F \rightarrow (E) \quad (7)$$

$$F \rightarrow i \quad (8)$$

Salida del programa.

```
tona@anti: ~/Documents/quinto/compiladores/practicas/sintactico/LL
File Edit View Search Terminal Help
→ LL git:(master) ✕ python prueba.py
Nombre del archivo: gramatica3.txt
Tabla LL(1):
  +   *   (   )   i   $
None None {1} None {1} None E
None None {4} None {4} None T
{2} None None {3} None {3} R
{6} {5} None {6} None {6} Y
None None {7} None {8} None F
→ LL git:(master) ✕
```

Figura 4: Tabla LL(1).

Esta tabla no produce ningún problema por lo que podría ser utilizada en el siguiente paso del análisis sintáctico.

4. Conclusiones

El uso de este tipo de analizadores resulta ser más poderosos que utilizar un analizador por descenso recursivo, sin embargo, aun existen problemas trabajando con gramáticas recursivas por la izquierda y con aquellas que pueden generar ambigüedades. Este es un serio problema debido a que a pesar de existir técnicas para eliminar la recursión por la izquierda no siempre es sencillo realizar este procedimiento por lo que al final optar por un tipo de analizador más poderoso puede reducir el tiempo de trabajo que se tenga que realizar.

Finalmente se puede concluir que de trabajar con alguna gramática sencilla utilizar este tipo de analizador seria lo indicado ya que no es difícil de entender ni de implementar.

Referencias

- [1] V. A. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1st ed., 1986.