

Reporte: Práctica 5

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

18 de noviembre de 2017

Índice

| | |
|-----------------|----|
| 1. Introducción | 2 |
| 2. Desarrollo | 3 |
| 3. Resultados | 8 |
| 4. Conclusiones | 10 |
| Referencias | 10 |

1. Introducción

Un analizador por descenso recursivo es un analizador sintáctico de arriba hacia abajo en el cual un conjunto de métodos recursivos son usados para procesar una entrada. El análisis de arriba hacia abajo puede ser visto como un intento de encontrar la derivación mas a la izquierda para una cadena. Esto genera que el árbol de sintaxis sea construido desde la raíz y creando nodos en preorden. [1]

En este tipo de analizador se tiene un método asociado a cada símbolo no terminal de la gramática.

2. Desarrollo

Se desarrollo un analizador sintáctico por descenso recursivo para un lenguaje bastante sencillo cuyas principales características son:

- Declaración de variables.
- Asignación de variables.
- Operaciones binarias (suma, resta, operaciones lógicas).
- Comparaciones (*if*).
- Ciclos (*while*).
- Fin de archivo con la palabra reservada *eof*.
- Fin de declaración de instrucción con punto y coma.

Es por esto que se tuvo que hacer un analizador léxico para reconocer este lenguaje, este analizador fue realizado utilizando la herramienta PLY. El código del analizador léxico es el siguiente.

```
1 tokens = [  
2     'p',  
3     'q',  
4     'c',  
5     'z',  
6     'a',  
7     'n',  
8 ]  
9 palabras_reservadas = {  
10     'if': 'i',  
11     'while': 'l',  
12     'var': 'v',  
13     'asig': 'b',  
14     'sumar': 'o',  
15     'mayorque': 'm',  
16     'restar': 'o',  
17     'menorque': 'm',  
18     'eof': 'f',  
19 }  
20 tokens += list(palabras_reservadas.values())  
21  
22 # Parentesis izq  
23 t_p = r'\(''  
24 # Parentesis der  
25 t_q = r'\)'  
26 # Coma  
27 t_c = r','
```

```

28 # Punto y coma
29 t_z = r';'
30
31
32 # ID
33 def t_a(t):
34     r'[a-zA-Z_][a-zA-Z_0-9]*'
35
36     # Check for reserved words
37     t.type = palabras_reservadas.get(t.value, 'a')
38     return t
39
40
41 # Numero
42 def t_n(t):
43     r'\d+'
44     t.value = int(t.value)
45     return t
46
47
48 def t_newline(t):
49     r'\n+'
50     t.lexer.lineno += len(t.value)
51
52
53 t_ignore = ' \t'
54
55
56 def t_error(t):
57     print("ERROR '%s'" % t.value[0])
58     t.lexer.skip(1)

```

Para el analizador sintáctico se definió la siguiente gramática con base a los tokens producidos por el analizador léxico.

$$\begin{aligned} A &\rightarrow vazA \\ A &\rightarrow I \\ i &\rightarrow opacacRqzI \\ R &\rightarrow a \\ R &\rightarrow n \\ C &\rightarrow mpacRq \\ I &\rightarrow bpacnqzI \\ I &\rightarrow ipCqI \\ I &\rightarrow lpCqI \\ I &\rightarrow f \\ I &\rightarrow czI \end{aligned}$$

Esta gramática se modelo en la siguiente clase y sus respectivos métodos.

```
1 import sys
2
3
4 class Analizador:
5     def __init__(self, token):
6         self.token = token
7         self.i = 0
8
9     def A(self):
10        if self.i == len(self.token):
11            print('No valida')
12            sys.exit()
13        if self.token[self.i] == 'v':
14            self.consumir('v')
15            self.consumir('a')
16            self.consumir('z')
17            self.A()
18        else:
19            self.I()
20
21    def R(self):
22        if self.i == len(self.token):
23            print('No valida')
24            sys.exit()
25        if self.token[self.i] == 'a':
26            self.consumir('a')
27        elif self.token[self.i] == 'n':
28            self.consumir('n')
29        else:
```

```

30         print('No valida')
31         sys.exit()
32
33     def C(self):
34         if self.i == len(self.token):
35             print('No valida')
36             sys.exit()
37         if self.token[self.i] == 'm':
38             self.consumir('m')
39             self.consumir('p')
40             self.consumir('a')
41             self.consumir('c')
42             self.R()
43             self.consumir('q')
44         else:
45             print('No valida')
46             sys.exit()
47
48     def I(self):
49         if self.i == len(self.token):
50             print('No valida')
51             sys.exit()
52         if self.token[self.i] == 'o':
53             self.consumir('o')
54             self.consumir('p')
55             self.consumir('a')
56             self.consumir('c')
57             self.consumir('a')
58             self.consumir('c')
59             self.R()
60             self.consumir('q')
61             self.consumir('z')
62             self.I()
63         elif self.token[self.i] == 'i':
64             self.consumir('i')
65             self.consumir('p')
66             self.C()
67             self.consumir('q')
68             self.I()
69         elif self.token[self.i] == 'l':
70             self.consumir('l')
71             self.consumir('p')
72             self.C()
73             self.consumir('q')
74             self.I()
75         elif self.token[self.i] == 'f':
76             self.consumir('f')
77         elif self.token[self.i] == 'b':

```

```
78         self.consumir('b')
79         self.consumir('p')
80         self.consumir('a')
81         self.consumir('c')
82         self.consumir('n')
83         self.consumir('q')
84         self.consumir('z')
85         self.I()
86     else:
87         self.C()
88         self.consumir('z')
89         self.I()
90
91     def consumir(self, simbolo):
92         if self.i == len(self.token):
93             print('No valida')
94             sys.exit()
95         if simbolo == self.token[self.i]:
96             self.i += 1
97         else:
98             print('No valida')
99             sys.exit()
```

3. Resultados

Para realizar las pruebas correspondientes se utilizo la siguiente clase en donde inicializamos el analizador léxico para que nos devuelva el grupo de tokens a procesar en el analizador sintáctico.

```
1 import reglas
2 from ply.lex import lex
3 from sintactico import Analizador
4
5
6 class Prueba:
7     def __init__(self):
8         # Iniciamos nuestro analizador lexico
9         self.lexer = lex(module=reglas)
10
11     def analizar(self, cadena):
12         entrada = ''
13         # Parte del analizador lexico
14         self.lexer.input(cadena)
15         for token in self.lexer:
16             entrada += token.type
17
18         print(entrada)
19         # Inicializamos el analizador sintactico
20         sintactico = Analizador(entrada)
21         resultado = sintactico.A()
22         if sintactico.i == len(sintactico.token):
23             print('Valida')
24         else:
25             print('No valida')
26
27 analizador = Prueba()
28 archivo = open('prueba.txt', 'r')
29 analizador.analizar(archivo.read())
```

En esta misma clase se introduce el archivo con el contenido a analizar, primero se introdujo el siguiente archivo el cual está bien estructurado por lo que nuestro analizador debería de indicar que el archivo es valido.

```
1 var hola;
2 var temp;
3
4 asig(id, 5);
5 sumar(id, id, 7);
6
7 mayorque(id, 8);
8
```

```

9  if (mayorque(id, i))
10     restar(id, id, 7);
11  while (menorque(id, 777))
12     sumar(id, id, id);
13
14  eof

```

Como se puede apreciar en la figura 1 el resultado del analizador fue el esperado.

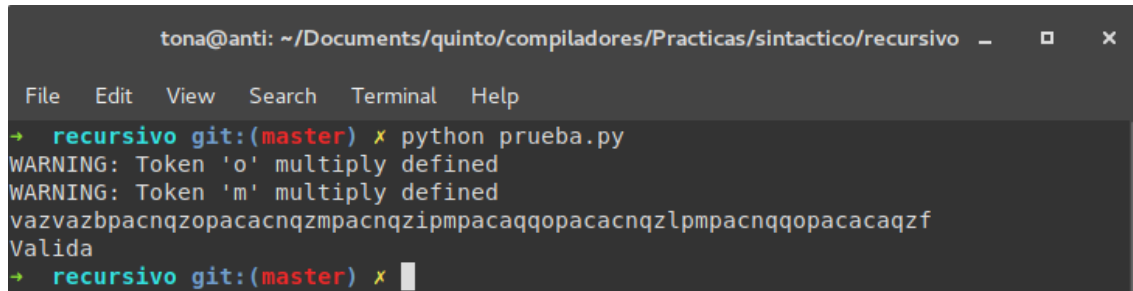


Figura 1: Prueba 1 sobre un archivo bien estructurado.

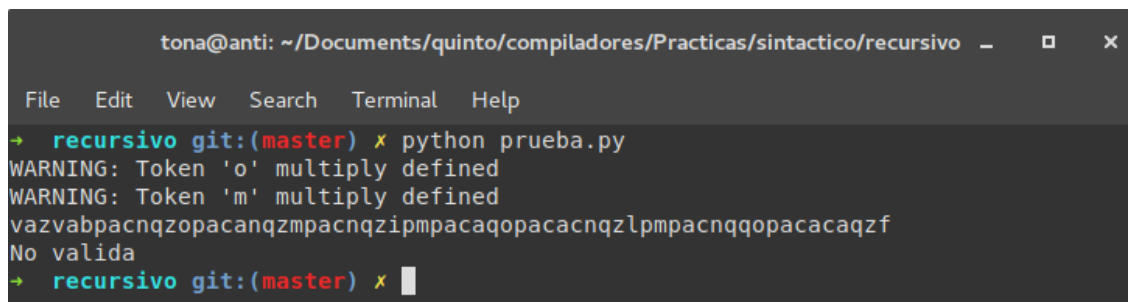
Ahora, el archivo que se introdujo en la prueba 1 se modifico para que presentara algunos errores por lo que el analizador sintáctico desarrollado debería de indicar que el archivo de entrada no es valido, cosa que sucede como se muestra en la figura ?? por lo que podemos concluir que funciona correctamente. El archivo modificado que se use esta vez fue el siguiente.

```

1  var hola;
2  var temp
3
4  asig(id, 5);
5  sumar(id, id 7);
6
7  mayorque(id, 8);
8
9  if (mayorque(id, i)
10     restar(id, id, 7);
11  while (menorque(id, 777))
12     sumar(id, id, id);
13
14  eof

```

En este archivo, se removieron símbolos como punto y coma, paréntesis y algunas comas por lo que el analizador sintáctico imprima que no es valido.



```
tona@anti: ~/Documents/quinto/compiladores/Practicas/sintactico/recursivo
File Edit View Search Terminal Help
→ recursivo git:(master) x python prueba.py
WARNING: Token 'o' multiply defined
WARNING: Token 'm' multiply defined
vazvabpacnqzopacnqzmpacnqzipmpacaqopacacnqzlpmpacnqqopacacaqzf
No valida
→ recursivo git:(master) x
```

Figura 2: Prueba 2 sobre un archivo con errores.

4. Conclusiones

El analizador sintáctico por descenso recursivo es bastante sencillo por lo que es fácil entender el como funciona es por esto que para ejemplos como el trabajado en esta practica es bastante útil debido a que la gramática también es simple, esto se ve reflejado en el hecho de que la parte más difícil de esta práctica fue definir una correcta gramática que permitiera modelar el lenguaje que se trabajo y el resto fue abstraer dicha gramática a nivel programación.

Sin embargo, al trabajar con ejemplos más complejos este tipo de analizador sintáctico no sera tan conveniente debido a las limitaciones que presenta para su correcto funcionamiento como lo es el hecho que una gramática que sea recursiva por la izquierda puede provocar que se entre en un bucle infinito debido a que si nos encontramos una producción que intente expandir un símbolo no terminal A al tener dicha recursión nos volveremos a encontrar con la situación de expandir otra vez a A . A pesar de esto el conocer como funciona permite entender el funcionamiento del resto de analizadores sintácticos y el porque se su existencia.

Referencias

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