UNIVERDIDAD TECNOLÓGICA NACIONAL

Facultad Regional Delta

IMPLEMENTACIÓN DE UN ANALIZADOR SINTÁCTICO

TRABAJO DE LABORATORIO N°2

MATERIA: Sintaxis y semántica de los lenguajes.

ALUMNOS: NUÑEZ OLMOS, German Imanol

RYSER, Mateo

CARRERA: Ingeniería en Sistemas de Información

DOCENTES: SANTOS, Juan Miguel

MIRANDA, Hernán

AÑO: 2019

Objetivo

Implementar un analizador sintáctico descendente para una gramática especificada.

Enunciado

La implementación del analizador sintáctico descendente con retroceso (ASDR) se realizará en grupo de hasta tres alumnos (preferentemente dos alumnos). Cada grupo recibirá una gramática para implementar el analizador. El tipo de ASDR deberá ser implementado mediante procedimientos, esto es, deberá haber un procedimiento Principal, un procedimiento Procesar y luego un procedimiento por cada no terminal de la gramática.

El programa que resulte de la implementación deberá aceptar una cadena y luego indicar si dicha cadena pertenece al lenguaje generado por la gramática y además deberá indicar qué producciones de la gramática deben ser usadas para derivar la cadena de entrada. Conjuntamente con la gramática, a cada grupo se le dará un conjunto de cadenas de testeo, de las cuales, algunas pertenecerán al lenguaje generado por la gramática y otras no.

Gramática Sintáctica

```
Programa → ListaDecl “eof”
ListaDecl → ListaDecl Declaracion | λ
Declaracion → FunDecl | VarDecl | Sentencia
FunDecl → "fun" Funcion
Funcion → Identificador "(" ListaParametros ")" Bloque
ListaParametros → λ | Parametros
Parametros → Identificador | Parametros "," Identificador
VarDecl → "var" Identificador ";" |"var" Identificador "=" Expresion ";"
Sentencia → ExprSent | ForSent | IfSent | ReturnSent | WhileSent | Bloque
ExprSent → Expresion ";"
Expresion → Asignacion
Asignacion → Identificador "=" Primitivo | OLogico;
ForSent → "for" "(" PriArg AdicArg ";" AdicArg ")" Sentencia
PriArg → VarDecl | ExprSent | ";"
AdicArg → λ | Expresion
IfSent → "if" "(" Expresion ")" Sentencia "else" Sentencia |
"if" "(" Expresion ")" Sentencia
ReturnSent → "return" Expresion ";" |"return" ";"
WhileSent → "while" "(" Expresion ")" Sentencia
Bloque → "{" ListaSent "}"
ListaSent → Sentencia ListaSent | λ
OLogico → YLogico | YLogico "or" OLogico
YLogico → Igua | Igua "and" YLogico
Igua → Comparacion | Comparacion "==" Igua | Comparacion "!=" Igua
Comparacion → Suma | Suma ">" Comparacion | Suma ">=" Comparacion |
Suma "<" Comparacion | Suma "<=" Comparacion
Suma → Mult | "-" Suma | "+" Suma
Mult → Unario | "/" Mult | "*" Mult
Unario → "!" Unario | "-" Unario | Primitivo
Primitivo → "true" | "false" | Numero | String | Identificador |
"("Expresion ")"
```

Gramática Léxica

```
Numero → ListaDigito | ListaDigito "." ListaDigito
ListaDigito → Digito | Digito ListaDigito
String → “’” ListaSimbolos “’”
Identificador → Letra | Letra ListaSimbolos
ListaSimbolos → Letra | Digito | Letra ListaSimbolos | Digito ListaSimbolos
Letra → “a” ... “z” | “A” ... “Z”
Digito → “0” ... “9” ;
```

Desarrollo

A rasgos generales, el esquema de un compilador comienza con un código fuente que es ingresado a un analizador lexicográfico (lexer) como se vio en el Trabajo Práctico de Laboratorio N°1. La salida producida en esta etapa es una cadena o lista de tokens que servirá como entrada para la implementación del analizador sintáctico (parser). Para este trabajo, se implementará un ***analizador sintáctico descendente recursivo (ASDR).***

En primer lugar, es menester declarar correctamente la gramática sintáctica que se utilizará en el proceso de parsing. Para nuestro programa, se decidió definir la gramática como un diccionario cuyas claves son los símbolos no terminales de la gramática, y cada una se vincula con una lista de listas que representan las derivaciones de estos símbolos. Los símbolos terminales de la gramática son los tokens.

El objetivo es que el parser diga si una cadena es producida por una gramática y, en caso afirmativo, cómo es la derivación de los no terminales, de izquierda a derecha, para llegar a tal resultado.

```
productionRules = {
    'Programa' : [['ListaDecl', "EOF"]],
    'ListaDecl' : [['ListaDecl2']],
    'ListaDecl2' : [['Declaracion', 'ListaDecl2'], [ ]],
    'Declaracion' : [['FunDecl'], ['VarDecl'], ['Sentencia']],
    'FunDecl' : [["FUN", 'Funcion']],
    'Funcion' : [["ID", "(", 'ListaParametros', ")", 'Bloque']],
    'ListaParametros' : [['Parametros'], [ ]], #
    'Parametros' : [["ID", 'Parametros2']],
    'Parametros2' : [[",", "ID", 'Parametros2'], [ ]],
    'VarDecl' : [["VAR", "ID", ";"], ["VAR", "ID", "ALLOP", 'Expresion', ";"]],
    'Sentencia' : [['ExprSent'], ['ForSent'], ['IfSent'], ['ReturnSent'], ['WhileSent'], ['Bloque']],
    'ExprSent' : [['Expresion', ";"]],
    'Expresion' : [['Asignacion']],
    'Asignacion' : [["ID", "ALLOP", 'Primitivo'], ['OLogico']],
    'ForSent' : ["FOR", "(", 'PriArg', 'AdicArg', ";", 'AdicArg', ")", 'Sentencia'],
    'PriArg' : [['VarDecl'], ['ExprSent'], [";"]],
    'AdicArg' : [['Expresion'], [ ]], #
    'IfSent' : [["IF", "(", 'Expresion', ")", 'Sentencia', "ELSE", 'Sentencia'], ["IF", "(", 'Expresion', ")", 'Sentencia']],
    'ReturnSent' : [["RETURN", 'Expresion', ";"], ["RETURN", ";"]],
    'WhileSent' : [["WHILE", "(", 'Expresion', ")", 'Sentencia']],
    'Bloque' : [["{", 'ListaDecl', "}"]],
    'OLogico' : [['YLogico'], ['YLogico', "LOGICOP", 'OLogico']],
    'YLogico' : [['Igua'], ['Igua', "LOGICOP", 'YLogico']],
    'Igua' : [['Comparacion'], ['Comparacion', "COMPOP", 'Igua']],
    'Comparacion' : [['Suma'], ['Suma', "COMPOP", 'Comparacion']],
    'Suma' : [['Mult'], ["ARITOP", 'Suma']],
    'Mult' : [['Unario'], ["ARITOP", 'Mult']],
    'Unario' : [["UNARY", 'Unario'], ['Primitivo']],
```

Definición real de la gramática en el programa.

Estructura del algoritmo.

La proceso de parsing hace uso de tres funciones: *parse ( ), pni (simboloNoTerminal), y procesar (parteDerecha)*. La funcion *parse* inicializa el programa con la instrucción *pni ('Programa').* Aquí arranca el proceso de analisis sintáctico de la cadena de entrada hasta alcanzar una condición de corte (como se trata una lista de tokens, la condición de corte es cuando se alcanza el tipo de token "*EOF"*). Cuando se cumple que:

if self['error'] == False and self['tokenInput'] == 'EOF':

Significa que la cadena fue aceptada por la gramática, por lo que el programa procede a mostrar el arbol de derivación para dicha cadena y retorna un valor *True* significando el resultado satifactorio.

El formato de impresión del arbol de derivación es:

" ", symbol, " => ", right, " "

```
def parse ():
    pni ('Programa')

    if self['error'] == False and self['tokenInput'] == 'EOF':
        print("ARBOL DE DERIVACION:")
        self['sentenceDiagram'].reverse()
        for (symbol, right) in self['sentenceDiagram']:
            print (" ", symbol, " => ", right, " ")
        return True
    else:
        return False
```

Definición de la función parse ( ).

Además, se implementa un diccionario dentro de la función parser ( ) con las variables más importantes, como la lista de tokens de entrada y el árbol de derivación que, durante el tiempo de ejecución, se irá formando.

```
self = {
    'listTokens' : [i[0] for i in tokens] ,
    'index' : 0 ,
    'error' : False ,
    'sentenceDiagram' : [ ],
    'tokenInput' : " "
}
```

Definición del diccionario.

Por otra parte, existen dos funciones importantes que distinguen los simbolos analizados entre terminales y no terminales:

- *isNonTerminal (symbol):*
- *isTerminal (symbol):*

```
    def pni (nonTerminalSymbol):
        for rightPart in productionRules[nonTerminalSymbol]:
            self['error'] = False
            i_pivote = self['index']
            process (rightPart)

            if self['error'] == False:
                self['sentenceDiagram'].append((nonTerminalSymbol, rightPart))
                break
            elif self['error'] == True:
                self['index'] = i_pivote
```

Definición de pni (simboloNoTerminal)

```
def process (rightPart):
        for s in rightPart:

            self['tokenInput'] = self['listTokens'][self['index']]

            if isTerminal (s):

                if s == self['tokenInput']:
                    self['index'] += 1
                else:
                    self['error'] = True
                    break

            elif isNonTerminal (s):

                pni (s)

                if self['error'] == True:
                    break
```

Definición de procesar (parteDerecha)

Ejemplos

A continuación, se mostrarán dos cadenas para realizar sus respectivos analisis sintáctico. Una de ellas será aceptada mientras que la otra no.

Caso 1:

Cadena de entrada: ***while (carter = reagan) return bush = 'bush' ;***

tokens: ***['WHILE', '(', 'ID', 'ALLOP', 'ID', ')', 'RETURN', 'ID', 'ALLOP', 'STRING', ';', 'EOF']***

ARBOL DE DERIVACION:

```
Programa  =>  ['ListaDecl', 'EOF']
ListaDecl  =>  ['ListaDecl2']
ListaDecl2  =>  ['Declaracion', 'ListaDecl2']
ListaDecl2  =>  []
Declaracion  =>  ['Sentencia']
Sentencia  =>  ['WhileSent']
WhileSent  =>  ['WHILE', '(', 'Expresion', ')', 'Sentencia']
Sentencia  =>  ['ReturnSent']
ReturnSent  =>  ['RETURN', 'Expresion', ';']
Expresion  =>  ['Asignacion']
Asignacion  =>  ['ID', 'ALLOP', 'Primitivo']
Primitivo  =>  ['STRING']
Expresion  =>  ['Asignacion']
Asignacion  =>  ['ID', 'ALLOP', 'Primitivo']
Primitivo  =>  ['ID']
```

El resultado del analisis sintáctico es ***True***, es decir, la cadena fue aceptada.

Caso 2:

Cadena de entrada: ***else bolsonaro > dilma if { } ;***

tokens: ***['ELSE', 'ID', 'COMPOP', 'ID', 'IF', '{', '}', ';', 'EOF']***

Esta es una cadena que no es producida por la gramática, por lo que el parser no devuelve un arbol de derivación y el resultado del análisis sintáctico es ***False.***