UNIVERDIDAD TECNOLÓGICA NACIONAL

Facultad Regional Delta

IMPLEMENTACIÓN DE UN ANALIZADOR LEXICOGRÁFICO

TRABAJO DE LABORATORIO N°1

MATERIA: Sintaxis y semántica de los lenguajes.

ALUMNOS: GIMÉNEZ, Leandro

NUÑEZ OLMOS, German Imanol

RYSER, Mateo

CARRERA: Ingeniería en Sistemas de Información

DOCENTES: SANTOS, Juan Miguel

MIRANDA, Hernán

AÑO: 2019

Objetivo

Implementar un analizador lexicográfico para una gramática especificada.

Enunciado

La implementación del analizador lexicográfico se realizará en grupos de 3 integrantes como máximo. Cada grupo recibirá una gramática para implementar el analizador. Se pretende desarrollar un AF por cada tipo de token y luego el analizador deberá ser implementado mediante un AFD que incluye a todos los AF construidos para cada token. El programa que resulte de la implementación deberá aceptar una cadena que representa código escrito en el lenguaje generado por la gramática provista. Este código, visto como una cadena de caracteres ASCII, deberá ser convertido a una cadena de tokens correspondiente a la gramática provista.

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

Como se pidió en el enunciado del trabajo, se quiere hacer una implementación de un analizador lexicográfico para la gramática dada.
El analizador lexicográfico comprende la primera parte de un compilador, que recibe un código fuente como entrada y devuelve como salida una lista de componentes léxicos denominados ***tokens***, que servirá de entrada para una siguiente parte del compilador, más precisamente, para el analizador sintáctico.

Para llevar a cabo la construcción del ***lexer*** (así llamaremos al analizador lexicográfico) se consideraron, primeramente, las gramáticas léxicas. Tomando las producciones de dichas gramáticas se construyeron los ***autómatas finitos*** que reconocen esa gramática:

```
def number_Automaton (string):
    state = 0
    final_states = [1,3]
    for c in string:
        if state == 0 and c.isdigit():
            state = 1
        elif state == 1 and c.isdigit():
            state = 1
        elif state == 1 and c == ".":
            state = 2
        elif state == 2 and c.isdigit():
            state = 3
        elif state == 3 and c.isdigit():
            state = 3
        else:
            state = TRAP
            break
    if state == TRAP:
        return TRAP_RESULT
    if state in final_states:
        return ACCEPT_RESULT
    else:
        if state != TRAP:
            return NOACCEPT_RESULT
```

*Ejemplo de un autómata para una gramática lexicográfica. En este caso, para **Numero**.*

Así, se construyeron los demás autómatas de las gramáticas en cuestión.

Posteriormente, se procedió a construir los autómatas finitos para las gramáticas sintácticas. En este caso, se crearon los autómatas para los terminales de cada una de las producciones de la gramática sintáctica:

```
def logicalOperators_Automaton(string):
    if string in ["or", "and", "not"]:
        return ACCEPT_RESULT
    else:
        return TRAP_RESULT

def arithmeticOperators_Automaton(string):
    state = 0
    final_states = [1, 2]
    for c in string:
        if state == 0 and c in ["+", "-", "*", "/", "%"]:
            state = 1
        elif state == 1 and c in ["*", "/"]:  #contemplo la posibilidad de exponente y division entera
            state = 2

        else:
            state = TRAP
            break

    if state == TRAP:
        return TRAP_RESULT
    if state in final_states:
        return ACCEPT_RESULT
    else:
        if state != TRAP:
            return NOACCEPT_RESULT
```

*En este caso se puede observar los autómatas para los operadores aritméticos y lógicos***.**

De forma semejante, se construyeron para palabras reservadas:

```
def if_Automaton(string):
    final_states = 2
    state = 0
    for c in string:
        if state == 0 and c == "i":
            state = 1
        elif state == 1 and c == "f":
            state = 2
        else:
            state = TRAP
            break

    if state == TRAP:
        return TRAP_RESULT

    if state == final_states:
        return ACCEPT_RESULT
    else:
        if state != TRAP:
            return NOACCEPT_RESULT
```

Aquí se muestra el autómata para la sentencia "if".

De esta forma se crearon todos los autómatas necesarios para implementar el lexer para la gramática dada. Pero antes de llevar a cabo una implementación, primero se debió definir una ***jerarquía de tokens***. Esto es para poder clasificar las cadenas de entrada y asignarles sus identificadores correctamente.

La jerarquía de tokens para este lexer se comprende de una lista formadas por tuplas o pares ordenados de la forma ***(funcion_automata, clave_token)***:

```
RANK_TOKENS = [
    (eof_Automaton, 'EOF'),
    (if_Automaton, 'IF'),
    (for_Automaton, 'FOR'),
    (while_Automaton, 'WHILE'),
    (else_Automaton, 'ELSE'),
    (return_Automaton, 'RETURN'),
    (fun_Automaton, 'FUN'),
    (var_Automaton, 'VAR'),
    (true_Automaton, 'TRUE'),
    (false_Automaton, 'FALSE'),
    (parOp_automaton, '('),
    (parCl_automaton, ')'),
    (marks_automaton, '"'),
    (semicon_automaton, ';'),
    (logicalOperators_Automaton, 'LOGICOP'),
    (allocationOperators_automaton, 'ALLOP'),
    (comparisonOperators_Automaton, 'COMPOP'),
    (arithmeticOperators_Automaton, 'ARITOP'),
    (string_Automaton, 'STRING'),
    (number_Automaton, 'NUMBER'),
    (ID_Automaton, 'ID')]
```

Lista de jerarquía de tokens real usada en el lexer.

Se entiende que, cuando una cadena no puede ser generada por la gramática dada, no se considera en la jerarquía de tokens, pero se verá que, en el programa principal, a esas cadenas se les asigna el clave ***ERROR_TOKEN***.

Finalmente se hace la implementación del lexer con el siguiente programa:

```
def lexer(src):
    tokens = []
    src = src + " "
    i = 0
    start = 0
    state = 0
    while i < len(src):
        caracter = src[i]
        lexeme = src[start:i+1]
        if caracter.isspace():
            i += 1
            start = i
        else:
            biggest_lexeme = src[start:i+1] #lo uso solo para el error
            ##evaluo segun cada automata y avanzo hasta q sean trampa
            while len(generateCandidates(lexeme)) != 0 and not caracter.isspace():
                i += 1
                state = 1
                lexeme = src[start:i+1]
                caracter = src[i]
                biggest_lexeme = lexeme
            i -= 1
            lexeme = src[start:i+1]
            while len(hasTrueCandidates(lexeme)) != 0 and len(lexeme) < 0:
                i -= 1
                lexeme = src[start:i+1]
            lista_candidatos = hasTrueCandidates(lexeme)
            i += 1
            if len(hasTrueCandidates(lexeme)) == 0: ##si la lista de candidatos aceptados es vacia, tiro error
                print("LEXEME NO RECONOCIDO EN POSICION "+str(i))
                tokens.append(('ERROR_TOKEN', biggest_lexeme))
                break
            else:   ##si tengo una lista de candidatos aceptados, tomo el primero
                token = lista_candidatos[0][0]
                tokens.append((token, lexeme))
            start = i ##marco el comienzo del lexeme nuevo
            state = 0
            biggest_lexeme = ''
    print("FIN")
    return(tokens)
```

Se definieron las siguientes funciones auxiliares para delegar tareas:

```
def generateCandidates(string):
    candidates = []
    for (automaton, token) in RANK_TOKENS:
        if automaton(string) != TRAP_RESULT:
            candidates.append((token, string,automaton(string)))
    #print(candidates)
    return candidates

def hasTrueCandidates(string):
    trueCandidates = []
    for (automaton, token) in RANK_TOKENS:
        if automaton(string) == ACCEPT_RESULT:
            trueCandidates.append((token,string))
    return trueCandidates
```

La funcion ***generateCandidates(string):*** toma una cadena y la ingresa en cada uno de los autómatas definidos al principio y que se encuentran en la jerarquía de tokens y genera una lista de tuplas de posibles tokens válidos para esa cadena.

La función ***hasTrueCandidates(string):*** es de funcionamiento similar a ***generateCandidates(string):*** con la diferencia que devuelve una lista de tuplas con el token y la cadena, cuando ésta última al ser ingresada a un autómata, se alcanza un estado aceptado.

Ejemplos de aplicación:

Se ingresan las siguientes cadenas:

- `"probando con un for"`
- `"'aloy' 12345 or"`
- `")("`
- `"musica_a_mi _ alrededor"`

Se espera que los resultados para ellas sean:

- `[('ID', 'probando'), ('ID', 'con'), ('ID', 'un'), ('FOR', 'for')]`
- `[('STRING', "'aloy'"), ('NUMBER','12345'),('LOGICOP', 'or')]`
- `[(')', ')'), ('(', '(')]`
- `[('ID', 'musica_a_mi'), ('ERROR_TOKEN', '_')]`

Si se ejecuta el programa se obtiene que:

```
FIN
FIN
FIN
LEXEME NO RECONOCIDO EN POSICION 12
FIN
Las pruebas del lexer
resultado 1:  [('ID', 'probando'), ('ID', 'con'), ('ID', 'un'), ('FOR', 'for')]
resultado 4:  [('STRING', "'aloy'"), ('NUMBER', '12345'), ('LOGICOP', 'or')]
resultado 9:  [(')', ')'), ('(', '(')]
resultado 10:  [('ID', 'musica_a_mi'), ('ERROR_TOKEN', '_')]

[Finished in 0.4s]
```

Se observa que los resultados se confirman. Se puede concluir que el lexer es funcional y cumple con las necesidades.