



ESCUELA POLITÉCNICA NACIONAL

Facultad de Ingeniería Software

Nombre: Sánchez Doménica

Fecha: 06/01/2023

COMPILADOR STARNFORD – PYTHON - LINUX

MV de Linux donde se encuentra nuestro compilador realizado en python

```
PrimeraMaquinaUbuntu [Corriendo] - Oracle VM VirtualBox
Archivo Máquina Ver Entrada Dispositivos Ayuda
ene 6 01:11
es
Actividades Terminal
camila@PrimeraMaquinaUbuntu: ~/Documents/Compilador-a...
camila@PrimeraMaquinaUbuntu:~$ cd
camila@PrimeraMaquinaUbuntu:~$ ls
Desktop Documents Downloads Music Pictures Public snap Templates Videos
camila@PrimeraMaquinaUbuntu:~$ cd Documents
camila@PrimeraMaquinaUbuntu:~/Documents$ ls
Compilador-analisis-lexer-Python PeachCompiler
camila@PrimeraMaquinaUbuntu:~/Documents$ cd Compilador-analisis-lexer-Python/
camila@PrimeraMaquinaUbuntu:~/Documents/Compilador-analisis-lexer-Python$ ls
analizador_lexico.py      main.py      pruebas      recursosGUI
analizador_sintactico.py  parsetab.py  __pycache__
camila@PrimeraMaquinaUbuntu:~/Documents/Compilador-analisis-lexer-Python$
```

En el desarrollo de este programa, se ha integrado los componentes esenciales de un compilador. Este incluye un analizador léxico, sintáctico y el parser correspondiente. Además, se diseñó una (GUI) destinada a facilitar el uso del compilador. Se incluyeron archivos específicos de pruebas.

MAIN

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import QFont
from PyQt5.uic.properties import QtGui
from PyQt5.QtWidgets import *
from PyQt5.uic.properties import QtGui

""" Importamos todas nuestras Ventana y funciones utiles"""
from recursoGUI.home import *
from analizador_lexico import *
from analizador_sintactico import *

class Main(QMainWindow):
    """ Clase principal de nuestra app"""
    def __init__(self):
        """ Inicializamos nuestra app"""
        QMainWindow.__init__(self)

        # Instanciamos nuestra ventana widget home
        self.home = Ui_home()
        self.home.setupUi(self)

        # Eventos
        self.home.bt_lexico.clicked.connect(self.ev_lexico)
        self.home.bt_sintactico.clicked.connect(self.ev_sintactico)
        self.home.bt_archivo.clicked.connect(self.ev_archivo)
        self.home.bt_limpiar.clicked.connect(self.ev_limpiar)

    def ev_lexico(self):
        """
        Manejo de analisis de expresion lexemas
        :return:
        """
```

```
def ev_lexico(self):
    """
    Manejo de analisis de expresion lexemas
    :return:
    """
    # print("lexico")

    # limpiamos el campo
    self.home.tx_lexico.setText('')

    #Obtenemos los datos ingresados
    datos = self.home.tx_ingreso.toPlainText().strip()

    # analizamos la lexemas de los datos ingresados
    resultado_lexico = prueba(datos)

    # self.home.tx_lexico.setText("Analizando lexico")
    cadena= ''
    for lex in resultado_lexico:
        cadena += lex + "\n"
    self.home.tx_lexico.setText(cadena)

def ev_sintactico(self):
    """
    Manejo de analisis gramatico
    :return:
    """
    # print("sintactico")

    # limpiamos el campo
    self.home.tx_sintactico.setText('')
    #Obtenemos los datos ingresados
    datos = self.home.tx_ingreso.toPlainText().strip()
```

```
    #Obtenemos los datos ingresados
    datos = self.home.tx_ingreso.toPlainText().strip()

    #analizamos la gramatica de los datos ingresados
    resultado_sintactico = prueba_sintactica(datos)
    cadena = ''

    #Armanos la cadena a mostrar
    for item in resultado_sintactico:
        cadena += item + "\n"
    # mostramos en pantalla
    self.home.tx_sintactico.setText( cadena )

def ev_archivo(self):
    """
    Manejo de subir archivo
    :return:
    """
    dlg = QFileDialog()

    if dlg.exec_():
        filenames = dlg.selectedFiles()
        f = open(filenames[0], 'r')

        with f:
            data = f.read().strip()
            if data:
                self.home.tx_ingreso.setText(data+"\n")

def ev_limpiar(self):
    """
    Manejo de limpieza de campos
    :return:
    """
    self.home.tx_ingreso.setText('')
```

```
def ev_limpiar(self):
    """
    Manejo de limpieza de campos
    :return:
    """
    self.home.tx_ingreso.setText('')
    self.home.tx_lexico.setText('')
    self.home.tx_sintactico.setText('')

def iniciar():
    # Instanciamos nuestro app por defecto esto no cambia
    app = QApplication(sys.argv)

    # Instanciamos nuestra ventana
    ventana = Main()
    # Mostramos nuestra app
    ventana.show()

    #Controlamos el cierre de la app
    sys.exit(app.exec_())

if __name__ == '__main__':
    iniciar()
```

El archivo main.py sirve como punto de entrada principal para la aplicación de análisis léxico y sintáctico. Importa los módulos necesarios, incluyendo PyQt5 para la interfaz gráfica y módulos personalizados como home, analizador_lexico y analizador_sintactico.

ANALIZADOR_LEXICO

El archivo analizador_lexico.py es responsable de implementar el analizador léxico, es la primera fase de un compilador y se encarga de convertir una secuencia de caracteres en una secuencia de tokens.

```
1. import ply.lex as lex

# resultado del analisis
resultado_lexema = []

reservada = (
    # Palabras Reservadas
    'INCLUDE',
    'USING',
    'NAMESPACE',
    'STD',
    'COUT',
    'CIN',
    'GET',
    'CADENA',
    'RETURN',
    'VOID',
    'INT',
    'ENDL',
)

tokens = reservada + (
    'IDENTIFICADOR',
    'ENTERO',
    'ASIGNAR',

    'SUMA',
    'RESTA',
    'MULT',
    'DIV',
    'POTENCIA',
    'MODULO',

    'MINUSMINUS',
    'PLUSPLUS',

2. #Condiciones
    'SI',
    'SINO',
    #Ciclos
    'MIENTRAS',
    'PARA',
    #logica
    'AND',
    'OR',
    'NOT',
    'MENORQUE',
    'MENORIGUAL',
    'MAYORQUE',
    'MAYORIGUAL',
    'IGUAL',
    'DISTINTO',
    # Symbolos
    'NUMERAL',

    'PARIZQ',
    'PARDER',
    'CORIZQ',
    'CORDER',
    'LLAIZQ',
    'LLADER',

    # Otros
    'PUNTOCOMA',
    'COMA',
    'COMDOB',
    'MAYORDER', #>>
    'MAYORIZQ', #<<
)
```

```
# Reglas de Expresiones Regulares para token de Contexto simpl

t_SUMA = r'\+'
t_RESTA = r'\-'
t_MINUSMINUS = r'\--\-'
# t_PUNTO = r'\.'
t_MULT = r'\*'
t_DIV = r'\/'
t_MODULO = r'\%'
t_POTENCIA = r'\{2} | \^'

t_ASIGNAR = r'='
# Expresiones Logicas
t_AND = r'\&\&'
t_OR = r'\||{2}'
t_NOT = r'\!'
t_MENORQUE = r'<'
t_MAYORQUE = r'>'
t_PUNTOCOMA = r';'
t_COMA = r','
t_PARIZQ = r'\('
t_PARDER = r'\)'
t_CORIZQ = r'\['
t_CORDER = r'\]'
t_LLAIZQ = r'\{'
t_LLADER = r'\}'
t_COMDOB = r'\''

def t_INCLUDE(t):
    r'include'
    return t

def t_USING(t):
```

```
    r'using'
    return t

def t_NAMESPACE(t):
    r'namespace'
    return t

def t_STD(t):
    r'std'
    return t

def t_COUT(t):
    r'cout'
    return t

def t_CIN(t):
    r'cin'
    return t

def t_GET(t):
    r'get'
    return t

def t_ENDL(t):
    r'endl'
    return t

def t_SINO(t):
    r'else'
    return t

def t_SI(t):
    r'if'
    return t
```

3.

4.

```
def t_MAYORIGUAL(t):
    r'>='
    return t

def t_IGUAL(t):
    r'=='
    return t

def t_MAYORDER(t):
    r'<<'
    return t

def t_MAYORIZQ(t):
    r'>>'
    return t

def t_DISTINTO(t):
    r'!='
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_comments(t):
    r'\/*(.\n)*?\*/'
    t.lexer.lineno += t.value.count('\n')
    print("Comentario de multiple linea")

def t_comments_ONELine(t):
    r'\/*(.)*\n'
    t.lexer.lineno += 1
    print("Comentario de una linea")
t_ignore = '\t'
```

5.

```
t_ignore = '\t'

def t_error( t):
    global resultado_lexema
    estado = """Token no valido en la linea {:4} Valor {:16} Posicion {:4}""".format(str(t.lineno), str(t.value),
                                                                                      str(t.lexpos))
    resultado_lexema.append(estado)
    t.lexer.skip(1)

# Prueba de ingreso
def prueba(data):
    global resultado_lexema

    analizador = lex.lex()
    analizador.input(data)

    resultado_lexema.clear()
    while True:
        tok = analizador.token()
        if not tok:
            break
        # print("lexema de "+tok.type+" valor "+tok.value+" linea "+tok.lineno)
        estado = "Linea {:4} Tipo {:16} Valor {:16} Posicion {:4}""".format(str(tok.lineno),str(tok.type) ,str(tok.value),
                                                                                      str(tok.lexpos) )
        resultado_lexema.append(estado)
    return resultado_lexema

# Instanciamos el analizador lexico
analizador = lex.lex()

if __name__ == '__main__':
    while True:
        data = input("Ingrese: ")
        prueba(data)
        print(resultado_lexema)
```

6.

PRUEBA ANALIZADOR_LEXICO

```
$ python analizador_lexico.py
ingrese: 123
['Linea 1 Tipo ENTERO Valor 123 Posicion 0 ']
ingrese: asd213
['Linea 1 Tipo IDENTIFICADOR Valor asd213 Posicion 0 ']
ingrese: 123 0 jsua 12s1ws13
['Linea 1 Tipo ENTERO Valor 123 Posicion 0 ', 'Linea 1 Tipo ENTERO Valor 0 Posicion 4 ', 'Linea 1 Tipo IDENTIFICADOR Valor jsua Posicion 6 ', 'Linea 1 Tipo ENTERO Valor 12 Posicion 11 ', 'Linea 1 Tipo IDENTIFICADOR Valor s1ws13 Posicion 13 ']
ingrese: {{
['Linea 1 Tipo LLAIZQ Valor { Posicion 0 ', 'Linea 1 Tipo LLAIZQ Valor { Posicion 1 ']
ingrese: #
['Linea 1 Tipo NUMERAL Valor # Posicion 0 ']
ingrese: 1*
['Linea 1 Tipo ENTERO Valor 1 Posicion 0 ', 'Linea 1 Tipo MULT Valor * Posicion 1 ', 'Linea 1 Tipo COMDOB Valor " Posicion 2 ']
ingrese: 123 0 jsua 12s1ws13
```

El analizador léxico muestra que el analizador distingue entre números y palabras, asignando tipos específicos a cada entrada según su estructura.

ANALIZADOR_SINTACTICO

```
1. import ply.yacc as yacc
from analizador_lexico import tokens
from analizador_lexico import analizador

# resultado del analisis
resultado_gramatica = []

precedence = (
    ('right', 'ASIGNAR'),
    ('left', 'SUMA', 'RESTA'),
    ('left', 'MULT', 'DIV'),
    ('right', 'UMINUS'),
)
nombres = {}

def p_declaracion_asignar(t):
    'declaracion : IDENTIFICADOR ASIGNAR expresion PUNTOCOMA'
    nombres[t[1]] = t[3]

def p_declaracion_expr(t):
    'declaracion : expresion'
    # print("Resultado: " + str(t[1]))
    t[0] = t[1]

def p_expresion_operaciones(t):
    '''
    expresion : expresion SUMA expresion
              | expresion RESTA expresion
              | expresion MULT expresion
              | expresion DIV expresion
              | expresion POTENCIA expresion
              | expresion MODULO expresion
    ...
    if t[2] == '+':
```

```
2. t[0] = t[1] + t[3]
elif t[2] == '-':
    t[0] = t[1] - t[3]
elif t[2] == '*':
    t[0] = t[1] * t[3]
elif t[2] == '/':
    t[0] = t[1] / t[3]
elif t[2] == '%':
    t[0] = t[1] % t[3]
elif t[2] == '**':
    i = t[3]
    t[0] = t[1]
    while i > 1:
        t[0] *= t[1]
        i -= 1

def p_expresion_uminus(t):
    'expresion : RESTA expresion %prec UMINUS'
    t[0] = -t[2]

def p_expresion_grupo(t):
    '''
    expresion : PARIZQ expresion PARDER
              | LLAIZQ expresion LLADER
              | CORIZQ expresion CORDER
    ...
    t[0] = t[2]
# sintactico de expresiones logicas
def p_expresion_logicas(t):
    '''
    expresion : expresion MENORQUE expresion
              | expresion MAYORQUE expresion
              | expresion MENORIGUAL expresion
              | expresion MAYORIGUAL expresion
              | expresion IGUAL expresion
```

```

| expression MAYORIGUAL expression
| expression IGUAL expression
| expression DISTINTO expression
| PARIZQ expression PARDER MENORQUE PARIZQ expression PARDER
| PARIZQ expression PARDER MAYORQUE PARIZQ expression PARDER
| PARIZQ expression PARDER MENORIGUAL PARIZQ expression PARDER
| PARIZQ expression PARDER MAYORIGUAL PARIZQ expression PARDER
| PARIZQ expression PARDER IGUAL PARIZQ expression PARDER
| PARIZQ expression PARDER DISTINTO PARIZQ expression PARDER
...
if t[2] == "<": t[0] = t[1] < t[3]
elif t[2] == ">": t[0] = t[1] > t[3]
elif t[2] == "<=": t[0] = t[1] <= t[3]
elif t[2] == ">=": t[0] = t[1] >= t[3]
elif t[2] == "==": t[0] = t[1] is t[3]
elif t[2] == "!=": t[0] = t[1] != t[3]
elif t[3] == "<":
    t[0] = t[2] < t[4]
elif t[2] == ">":
    t[0] = t[2] > t[4]
elif t[3] == "<=":
    t[0] = t[2] <= t[4]
elif t[3] == ">=":
    t[0] = t[2] >= t[4]
elif t[3] == "==":
    t[0] = t[2] is t[4]
elif t[3] == "!=":
    t[0] = t[2] != t[4]

# print('logica ', [x for x in t])

# gramatica de expresiones booleanadas
def p_expression_booleana(t):
    ...
    expression : expression AND expression

```

3.

```

# gramatica de expresiones booleanadas
def p_expression_booleana(t):
    ...
    expression : expression AND expression
| expression OR expression
| expression NOT expression
| PARIZQ expression AND expression PARDER
| PARIZQ expression OR expression PARDER
| PARIZQ expression NOT expression PARDER
...
if t[2] == "&&":
    t[0] = t[1] and t[3]
elif t[2] == "||":
    t[0] = t[1] or t[3]
elif t[2] == "!":
    t[0] = t[1] is not t[3]
elif t[3] == "&&":
    t[0] = t[2] and t[4]
elif t[3] == "||":
    t[0] = t[2] or t[4]
elif t[3] == "!":
    t[0] = t[2] is not t[4]

def p_expression_numero(t):
    'expression : ENTERO'
    t[0] = t[1]

def p_expression_cadena(t):
    'expression : COMDOB expression COMDOB'
    t[0] = t[2]

```

4. def p_expression_nombre(t):

```

'expression : IDENTIFICADOR'
try:
    t[0] = nombres[t[1]]
except LookupError:
    print("Nombre desconocido ", t[1])
    t[0] = 0

def p_error(t):
    global resultado_gramatica
    if t:
        resultado = "Error sintactico de tipo {} en el valor {}".format(str(t.type),
str(t.value))
        print(resultado)
    else:
        resultado = "Error sintactico {}".format(t)
        print(resultado)
        resultado_gramatica.append(resultado)

# instanciamos el analizador sistactico
parser = yacc.yacc()

def prueba_sintactica(data):
    global resultado_gramatica
    resultado_gramatica.clear()

    for item in data.splitlines():
        if item:
            gram = parser.parse(item)
            if gram:
                resultado_gramatica.append(str(gram))
            else: print("data vacia")

print("result: ", resultado_gramatica)

```

5.

```

        resultado = "Error sintactico {}".format(t)
        print(resultado)
        resultado_gramatica.append(resultado)

# instanciamos el analizador sistactico
parser = yacc.yacc()

def prueba_sintactica(data):
    global resultado_gramatica
    resultado_gramatica.clear()

    for item in data.splitlines():
        if item:
            gram = parser.parse(item)
            if gram:
                resultado_gramatica.append(str(gram))
            else: print("data vacia")

    print("result: ", resultado_gramatica)
    return resultado_gramatica

if __name__ == '__main__':
    while True:
        try:
            s = input('ingresa dato >>> ')
        except EOFError:
            continue
        if not s: continue

        # gram = parser.parse(s)
        # print("Resultado ", gram)

    prueba_sintactica(s)

```

6.

[illegible]

```

ingresa dato >>> 123
result: ['123']
ingresa dato >>> qwe32
Nombre desconocido qwe32
result: []
ingresa dato >>> {21
Error sintactico None
result: ['Error sintactico None']
ingresa dato >>> 1+'31
result: ['32']
ingresa dato >>> 34242
result: ['34242']
ingresa dato >>> asads+123
Nombre desconocido asads
result: ['123']
ingresa dato >>>

```

Analizador_sintactico.py es un script en Python que define las reglas gramaticales y la lógica para analizar la estructura sintáctica de un lenguaje o datos de entrada.

PARSEAR

[illegible][illegible]

1.

2

El archivo `parsetab.py` es generado por `PLY`, una herramienta para Python que implementa `lex` y `yacc`. Contiene la tabla de análisis sintáctico usada por `PLY` para interpretar entradas basadas en reglas gramaticales del analizador. Esta tabla representa la gramática en una forma que permite el análisis y la construcción del árbol sintáctico correspondiente.

COMPILADOR CON GUI

Para que sea entendible por el usuario se incorporó una GUI y el compilador es:

