

Universidad San Carlos de Guatemala
Organización de Lenguajes y Compiladores 2
Escuela de Ciencias y Sistemas

Manual Usuario: Proyecto 1

Gary Joan Ortiz Lopez
200915609

INDICE

OBJETIVO GENERAL.....	3
OBJETIVO ESPECIFICO	3
Introducción	4
REQUISITOS MINIMOS	5
APLICACIÓN.....	¡Error! Marcador no definido.
COMPONENTES	¡Error! Marcador no definido.
REPORTES	¡Error! Marcador no definido.

OBJETIVO GENERAL

Desarrollar un intérprete en lenguaje Python el cual reconocerá instrucción de simples parece a código de 3 direcciones en php para ellos utilizando herramienta para graficar como GRAPHVIZ y herramientas para el analizar sintáctico y léxico como lo es PLY de Python.

OBJETIVO ESPECIFICO

- Ejecutar lenguaje de alto nivel con la ayuda de herramientas especializadas es el análisis léxico y sintáctico.
- Generar reportes en donde se muestre el estado de la ejecución.
- Visualizar el estado de la ejecución en la consola.

Introducción

Augus es un lenguaje de programación, basado en PHP y en MIPS. Su principal funcionalidad es ser un lenguaje intermedio, ni de alto nivel como PHP ni de bajo nivel como el lenguaje ensamblador de MIPS.

El lenguaje tiene dos restricciones: la primera, es que cada instrucción es una operación simple; y la segunda, es que en cada instrucción hay un máximo de dos operandos y su asignación (si la hubiera).

Mediante Compiladores 2 se muestra la capacidad de análisis obtenida por el estudiante ya que desarrollar una aplicación desde cero no suele ser tarea sencilla, esto por la interacción de componentes tales como:

- Componentes léxicos
- Componentes sintácticos (se relaciona con los componentes léxicos)
- Acciones a los componentes sintácticos.

Dicho todo esto, pasaremos a la descripción de la solución esperada para la entrega del primer proyecto de laboratorio

REQUISITOS MINIMOS

- Sistema operativo: Windows 8 o superior
- Python en su versión 3.6.5
- Ply (PHYTON LEX-YACC) 3.11
- Graphiz 2.38
- Memoria RAM 4Gb
- Disco Duro 512 Mb
- Visual Code o PYcharm para programación
- Paquete fpdf para generar los pdf

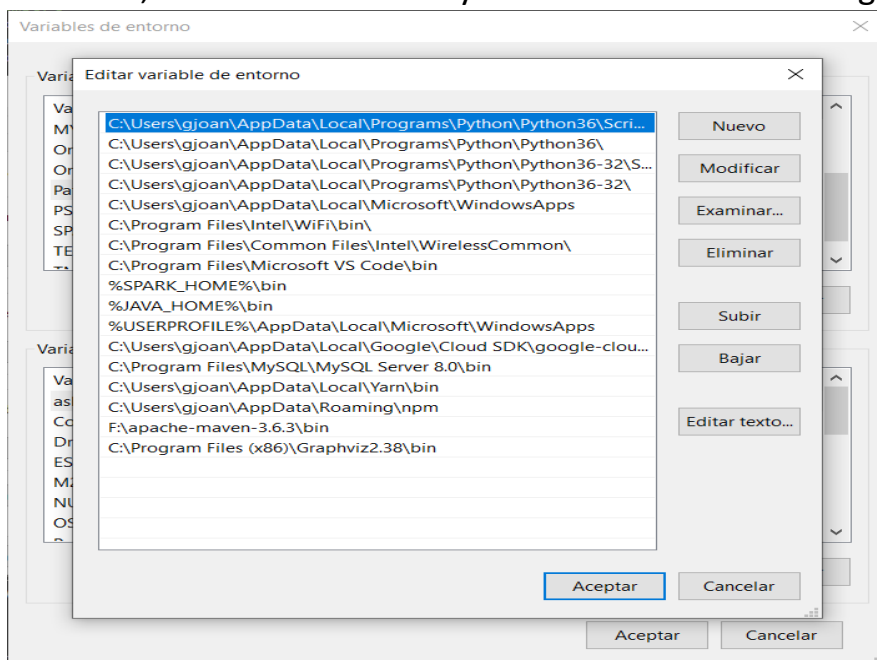
El contenido del código se encuentra almacenada en control de versiones:

Link:

https://github.com/Gary-Joan/Proyecto1_Compi2

Variables de entorno

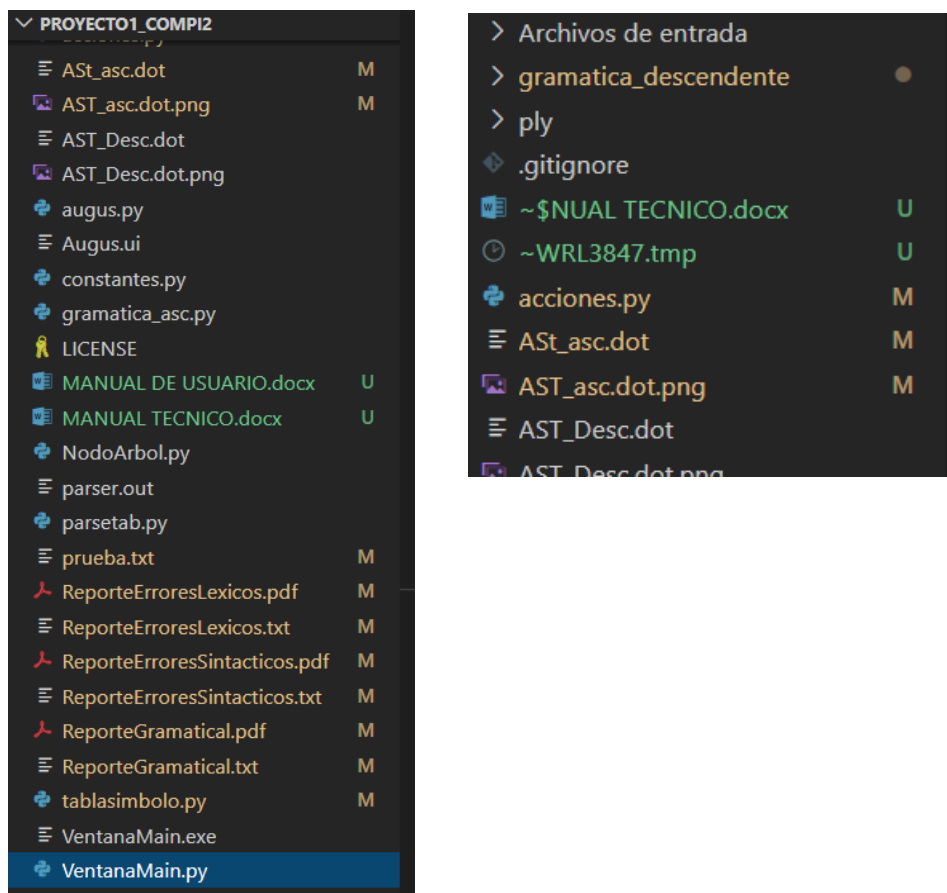
Para poder generar graficas se necesita tener instalado graphviz en el menú Inicio hacemos clic derecho a Mi Pc, seleccionamos la opción de “Propiedades”, luego seleccionamos la pestaña “Configuración Avanzada del sistema”, luego seleccionamos la opción de “Variables de entorno”, luego procedemos crear dentro del apartado de “Variables para del Usuario”, un CLASSPATH y un PATH con el siguiente formato:



Agregar al final de las variables de entorno el path donde tengamos instalado Graphviz

DIRECTORIO DE ARCHIVOS

Dentro de la aplicación se mostrar la estructura de archivos que se usaron para la aplicación



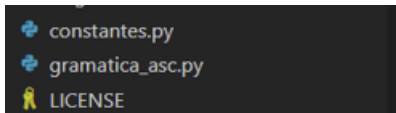
Directorio:

Aquí se muestra lo que contiene cada carpeta.

- Carpeta Ply: aquí se encuentra las herramientas que se usan para crear los analizadores léxicos y sintácticos

- Archivos de entrada: aquí se encuentran algunos archivos de prueba para probar la funcionalidad de la aplicación.
- Gramática_descendente: aquí se encuentra la parte del analizador descendente ya que se tienen que trabajar por separado para poder ejecutar ambas gramaticas.

A continuación, pasamos a explicar los archivos dentro la carpeta del proyecto.



El archivo **constantes.py** se encuentran variables constantes que sirven para llevar control de constantes con un valor para poder usarlas dentro de la aplicación.

Gramática_asc.py en este se encuentra la gramática para un analizar ascedenten el cual manera la herramienta PLY de Python

De muestra la estructura del archivo


```

palabrasreservadas = {
    'main' : 'MAIN',
    'print' : 'IMPRIMIR',
    'unset' : 'UNSET',
    'goto' : 'GOTO',
    'abs' : 'ABSOLUTO',
    'if' : 'IF',
    'read' : 'READ',
    'exit' : 'EXIT',
    'int' : 'INT',
    'float' : 'FLOAT',
    'char' : 'CHAR',
    'array' : 'ARRAY',
    'xor' : 'XOR'
}

tokens = [
    'DOSPUNTOS',
    'PARIZQ',
    'PARDER',
    'IGUAL',
    'PUNTERO',
    'COMENT',
    'PUNTOCOMA',
    'SUMA',
    'RESTA',

```

Aquí se muestran algunas palabras reservadas dentro de la aplicación así como los token utilizados para en análisis sintáctico.

A continuación, se muestra un fragmento de la gramática y como debe ser usada para poder generar el analizador

```

def p_init(t) :
    'inicio          : instruccion'
    print("Todo correcto!")
    constantes.reporte_gramatical.append(str(t.slice[0])+ " -> " +str(t.slice[1])+ "\n{ t[0]= t[1] }")
    t[0] = t[1]

    Raiz = t[0]

    recorrer_arbol(t[0])

def p_instruccion(t) :
    '''instruccion    : MAIN DOSPUNTOS listainstrucciones '''
    constantes.reporte_gramatical.append("instruccion ->" +str(t.slice[1].type)+ " "+str(t.slice[2].type)+str(t.slice[3].type)+ "\n{ t[0]=t[3] }")
    t[0]=t[3]

def p_listainstrucciones(t):
    'listainstrucciones : listainstrucciones lista'
    constantes.reporte_gramatical.append(str(t.slice[0].type)+ " -> " +str(t.slice[1].type)+ "\n{t[0]=t[1]; T[0]=agregar_hijo(t[0],t[2]) } " )

    t[0] = t[1]
    t[0] = agregar_hijo(t[0],t[2])

```

Dentro del archivo de compilación se encuentran los métodos para ir creando el árbol AST para después usado en con sus acciones semánticas.


```
def incrementar():
    constantes.numero+=1
    return constantes.numero

def crear_hoja(produccion, parametro):
    nodo = NodoArbol(produccion,parametro)
    return nodo



def agregar_hijo(nodo, hijo):
    nodo.agregar_hijos(hijo)
    return nodo
```

Uno crea una hoja para el árbol y el otro los une a su su padre para poder recorrerlo de mejor manera.

Se mostrará los archivos que manejar la lógica de las acciones y la construcción del árbol.

 NodoArbol.py

Este archivo contiene en nodo el cual será usado para crear el árbol.

```
NodoArbol.py >  NodoArbol >  __init__
class NodoArbol():
    def __init__(self,produccion, valor):
        self.produccion= produccion
        self.valor = valor
        self.hijos = []
    def agregar_hijos(self,Nodo):
        self.hijos.append(Nodo)
```

Aquí se agregan los hijos al árbol y se unirá con la creación de hojas del árbol.

TABLA DE SIMBOLOS



Aquí manejamos la tabla de símbolos que se en las acciones dentro de la aplicación.

```
class tablasimbolos(object):
    def __init__(self, simbolos={}):
        self.simbolos = simbolos

    def add_symbol(self, simbolo):
        self.simbolos[simbolo.id] = simbolo

    def get_symbol(self, id):
        if not id in self.simbolos:
            #print('Error: variable ', id, ' no definida.')
            return None
        else:
            return self.simbolos[id]

    def update_symbol(self, simbolo):
        if not simbolo.id in self.simbolos:
            print('Error: variable ', simbolo.id, ' no definida.')
        else:
            self.simbolos[simbolo.id] = simbolo

    def delete_symbol(self, simbolo):
        if not simbolo.id in self.simbolos:
            print('Error: variable', simbolo.id, 'no encontrada')
        else:
            self.simbolos.pop(simbolo.id)
```

Posee los métodos de agregar un símbolo, buscar un símbolo, darle una actualización a un símbolo y también poder eliminar un símbolo dentro de la tabla, dentro de este archivo se encuentra la inicialización del símbolo con todo los valores que va a poseer a continuación se muestran.

```

class TIPO_DATO(Enum):
    NUMERO = 1
    CADENA = 2
    BOOL = 3
    ARREGLO = 4

class Simbolo():
    # tipo puede ser entereo , decimal , todos los tipod de datos primitivos
    # id el nombre de la variable
    # valor si es un valor puntual guardameos el valor , si es vector creo una lista [ ]
    # rol nos indica si es una variable . si es un vec
    # tamano
    #
    def __init__(self,id,tipo,valor,rol,tamano,dim):
        self.id = id
        self.tipo = tipo
        self.valor = valor
        self.rol = rol
        self.tamano=tamano
        self.dim=dim
        #self.declarado = declarado
        #self.dimension = dimension
        #self.referencia = referencia

```

ARCHIVO ACCIONES

 acciones.py M

Aquí se encuentra las acciones que se ejecutaran leyendo el resultado el árbol que se genera dentro del analizador lexico, sintactico.

Aquí podemos ver algunos ejemplos

```

def ejecutar (self, consola):
    consola.setPlainText('')
    self.acciones(self.Raiz, consola)

def acciones(self, Raiz, consola):
    resutl=None
    consola1=consola
    if(Raiz!=None):
        if Raiz.produccion=='lista_inst':
            # listas de hijos de produccion lista de instancias
            self.lsen=Raiz.hijos # es la lista de hijos de lista_inta
            self.fin=len(self.lsen)
            for j in range(self.i, self.fin):
                self.posLabel+=1
                node=self.lsen[j]
                if(isinstance(node, LexToken)):
                    s=0
                else:
                    resutl=self.acciones(node, consola1)

```

Aquí podemos observar que recibe como parámetro la raíz del árbol generado y después pasa a verificar cada nodo del árbol y ejecuta las instrucciones dependiendo de la producción que venga.

GRAMATICAS

GRAMATICA ASCENDENTE

Aquí se muestra la gramática generado por el analizador el cual usa para poder hacer el análisis del archivo que se envíe.

- Rule 0 S' -> inicio**
- Rule 1 inicio -> instruccion**
- Rule 2 instruccion -> MAIN DOSPUNTOS listainstrucciones**
- Rule 3 listainstrucciones -> listainstrucciones lista**
- Rule 4 listainstrucciones -> lista**
- Rule 5 lista -> inst_asignacion**
- Rule 6 lista -> inst_imprimir**
- Rule 7 lista -> inst_if**
- Rule 8 lista -> inst_goto**
- Rule 9 lista -> etiqueta**
- Rule 10 lista -> inst_unset**
- Rule 11 lista -> inst_exit**
- Rule 12 lista -> error**
- Rule 13 inst_asignacion -> variable IGUAL expresion PUNTOCOMA**
- Rule 14 variable -> VAR**
- Rule 15 variable -> variable var_arreglo**
- Rule 16 var_arreglo -> LLAVEIZQ valorp LLAVEDER**
- Rule 17 expresion -> expresion_num**
- Rule 18 expresion -> conversion**
- Rule 19 expresion -> leer_valor**
- Rule 20 expresion -> variable**
- Rule 21 expresion -> inst_array**
- Rule 22 expresion_num -> valorp op valorp**
- Rule 23 op -> SUMA**
- Rule 24 op -> RESTA**

Rule 25 op -> MULTI

Rule 26 op -> DIV

Rule 27 op -> RESIDUO

Rule 28 op -> MAYORQUE

Rule 29 op -> MENORQUE

Rule 30 op -> IGUALQUE

Rule 31 op -> MAYORIGUALQUE

Rule 32 op -> MENORIGUALQUE

Rule 33 op -> NIGUALQUE

Rule 34 op -> AND

Rule 35 op -> OR

Rule 36 op -> XOR

Rule 37 op -> NOTBIT

Rule 38 op -> ANDBIT

Rule 39 op -> ORBIT

Rule 40 op -> XORBIT

Rule 41 expresion_num -> RESTA valorp

Rule 42 expresion_num -> ABSOLUTO PARIZQ valorp PARDER

Rule 43 expresion_num -> valorp

Rule 44 valorp -> DECIMAL

Rule 45 valorp -> ENTERO

Rule 46 valorp -> CADENA

Rule 47 valorp -> STRING

Rule 48 valorp -> CADENADOBLE

Rule 49 valorp -> VAR

Rule 50 valorp -> ID

Rule 51 expresion_num -> NOT valorp

Rule 52 conversion -> PARIZQ valor_conversion PARDER VAR

Rule 53 valor_conversion -> INT

Rule 54 valor_conversion -> FLOAT
Rule 55 valor_conversion -> CHAR
Rule 56 leer_valor -> READ PARIZQ PARDER
Rule 57 etiqueta -> ID DOSPUNTOS
Rule 58 inst_unset -> UNSET PARIZQ VAR PARDER PUNTOCOMA
Rule 59 inst_exit -> EXIT PUNTOCOMA
Rule 60 inst_array -> ARRAY PARIZQ PARDER
Rule 61 inst_imprimir -> IMPRIMIR PARIZQ expresion PARDER PUNTOCOMA
Rule 62 inst_if -> IF PARIZQ expresion PARDER GOTO ID PUNTOCOMA
Rule 63 inst_goto -> GOTO ID PUNTOCOMA

GRAMATICA DESCENDENTE

No existe mayor cambio una de la otra solo la eliminación de la recursividad por la izquierda para poder hacer que sea una gramática descendente.

Rule 0 S' -> inicio

Rule 1 inicio -> instruccion

Rule 2 instruccion -> MAIN DOSPUNTOS listainstrucciones

Rule 3 listainstrucciones -> lista listainstrucciones_prima

Rule 4 listainstrucciones_prima -> lista listainstrucciones_prima

Rule 5 listainstrucciones_prima -> <empty>

Aquí se quitó la recursividad para poder manejar dicha gramática

Rule 6 lista -> inst_asignacion

Rule 7 lista -> inst_imprimir

Rule 8 lista -> inst_if

Rule 9 lista -> inst_goto

Rule 10 lista -> etiqueta

Rule 11 lista -> inst_unset

Rule 12 lista -> inst_exit

Rule 13 lista -> error

Rule 14 inst_asignacion -> variable IGUAL expresion PUNTOCOMA

Rule 15 variable -> VAR variable_prima

Rule 16 variable_prima -> var_arreglo variable_prima

Rule 17 variable_prima -> <empty>

Aquí se quitó la recursividad para poder manejar dicha gramática

Rule 18 var_arreglo -> LLAVEIZQ valorp LLAVEDER

Rule 19 expresion -> expresion_num

Rule 20 expresion -> conversion

Rule 21 expresion -> leer_valor

Rule 22 expresion -> variable

Rule 23 expresion -> inst_array

Rule 24 expresion_num -> valorp op valorp

Rule 25 op -> SUMA

Rule 26 op -> RESTA

Rule 27 op -> MULTI
Rule 28 op -> DIV
Rule 29 op -> RESIDUO
Rule 30 op -> MAYORQUE
Rule 31 op -> MENORQUE
Rule 32 op -> IGUALQUE
Rule 33 op -> MAYORIGUALQUE
Rule 34 op -> MENORIGUALQUE
Rule 35 op -> NIGUALQUE
Rule 36 op -> AND
Rule 37 op -> OR
Rule 38 op -> XOR
Rule 39 op -> NOTBIT
Rule 40 op -> ANDBIT
Rule 41 op -> ORBIT
Rule 42 op -> XORBIT
Rule 43 expresion_num -> RESTA valorp
Rule 44 expresion_num -> ABSOLUTO PARIZQ valorp PARDER
Rule 45 expresion_num -> valorp
Rule 46 valorp -> DECIMAL
Rule 47 valorp -> ENTERO
Rule 48 valorp -> CADENA
Rule 49 valorp -> STRING
Rule 50 valorp -> CADENADOBLE
Rule 51 valorp -> variable
Rule 52 valorp -> ID
Rule 53 expresion_num -> NOT valorp
Rule 54 conversion -> PARIZQ valor_conversion PARDER VAR
Rule 55 valor_conversion -> INT

Rule 56 valor_conversion -> FLOAT
 Rule 57 valor_conversion -> CHAR
 Rule 58 leer_valor -> READ PARIZQ PARDER
 Rule 59 etiqueta -> ID DOSPUNTOS
 Rule 60 inst_unset -> UNSET PARIZQ VAR PARDER PUNTOCOMA
 Rule 61 inst_exit -> EXIT PUNTOCOMA
 Rule 62 inst_array -> ARRAY PARIZQ PARDER
 Rule 63 inst_imprimir -> IMPRIMIR PARIZQ expresion PARDER PUNTOCOMA
 Rule 64 inst_if -> IF PARIZQ expresion PARDER GOTO ID PUNTOCOMA
 Rule 65 inst_goto -> GOTO ID PUNTOCOMA

DATOS PERSONALES

Nombre	Carne	Telefono	Email
Gary Joan Ortiz Lopez	200915609	45961710	Gortiz1490@gmail.com