MANUAL TECNICO

AUGUS

15 de Junio de 2020

Sergio Geovany Guoz Tubac
201503925
ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 2

Contenido

Augus Application

Descripción

IDE de Augus es una aplicación con interfaz que funciona como un intérprete del lenguaje Augus. Un lenguaje intermedio creado especialmente para el curso de Organización de Lenguajes y Compiladores 2.

Para realizar el interprete se utilizó PLY de Python. Tal como se desarrollará a continuación.

Desarrollado con las tecnologías:

Sistema Operativo: Ubuntu 16.04 LTS

Lenguaje de Programación: Python

Versión de Python: 3.5.4

GUI: Tkinter

IDE: Visual Studio Code

Gráficas: graphviz PyPI

Construcción del Analizador

PLY: es una herramienta de análisis que está escrita en el lenguaje de programación Python. Es un generador de analizadores y es una implementación de lex y yacc. Utiliza el análisis LALR(1) que lo hace muy potente para gramáticas de todo tipo.

Para este proyecto se implementa una gramática ascendente y una descendente simulada, de la cual se detallará más adelante.

Estructura de PLY

- Ply.lex para realizar el análisis léxico.
- Ply.yacc para la creación de analizadores.

Definición del Analizador Léxico

Para realizar el analizador léxico se crean una serie de tokens y palabras reservadas que serán reconocidas por el analizador.

Cualquier otro carácter o expresión no registrada en la definición del analizador, dará como resultado un error y se guardará el error en una lista de errores.

Tokens

Se definen de la siguiente manera. Estos son secuencias de caracteres que serán de utilidad para el análisis sintactico.

Hay dos formas de reconocer tokens.

Primera forma de Reconocer los tokens

Se describen como una lista en la que cada identificador debe iniciar con **t_** y seguido se le asigna un valor que debe reconocer. La forma del token para ser reconocido por ply debe ser precedido del carácter r. Tal como se muestra a continuación:

```
t_TEMPVAR   = r'[$]t\d+'
t_PARAM     = r'[$]a\d+'
t_FUNVAL    = r'[$]v\d+'
```

Segunda forma de reconocer los tokens

Se realiza una función que reconocerá una secuencia de caracteres específica, definidos por una expresión regular, con la diferencia de que se castea valores, y se eliminan comillas.

```
def t_CADENA(t):
    r'\'.*?\''
    t.value = t.value[1:-1]
    return t
```

Lista de tokens para reconocer en la gramática de Augus.

```
# Tokens
t_MAS       = r'\+'
t_MENOS     = r'-'
t_MULTI     = r'\*'
t_DIVISION  = r'/'
t_RESIDUO   = r'%'
t_IGUAL     = r'='
t_PARIZQ    = r'\('
t_PARDER    = r'\)'
t_PTCOMA    = r';'
t_NOT       = r'!'
t_AND       = r'&&'
t_OR        = r'\|\|'
```

Palabras Reservadas

Palabras propias del lenguaje Augus son definidas de la siguiente manera, se adapta a un diccionario con información clave, valor. Estos tokens podemos utilizarlos en la definición del analizador sintáctico.

```
reservadas = {
    'goto'  : 'GOTO',
    'unset' : 'UNSET',
    'print' : 'PRINT',
    'read'  : 'READ',
    'exit'  : 'EXIT',
    'int'   : 'INT',
    'float' : 'FLOAT',
    'char'  : 'CHAR',
    'abs'   : 'ABS',
    'xor'   : 'XOR',
    'array' : 'ARRAY',
    'if'    : 'IF'
```

Manejo de Errores Léxicos

Una función de error para manejar los errores léxicos y es donde también se guarda el error en una lista para ser mostrado en el reporte de errores que es generado en la aplicación.

Un error léxico se dispara cuando el o los caracteres no concuerdan con ningún patrón de nuestras expresiones regulares que estamos definiendo.

t.lexer.skip(1) descarta el token.

t.lexer.lineno obtiene el número de línea del token.

```
def t_error(t):
    print("Caracter NO Valido: '%s'" % t.value[0])
    t.lexer.skip(1)
    nErr=ErrorRep('Lexico','Caracter NO Valido %s' % t.value[0],t.lexer.lineno)
    lisErr.agregar(nErr)
```

Construir el Analizador Léxico

Este analizador es el enviado para realizar el análisis sintáctico.

lexer2 es lo que le enviaremos al analizador sintáctico para poder realizar el análisis de los archivos de entrada.

```
#----------------------
import ply.lex as lex
lexer2 = lex.lex()
#----------------------
```

Construcción del Analizador Sintáctico

Patrón Interprete

Para construir el analizador sintáctico se hará uso del patrón interprete que es mucha utilidad para representar la gramática y realizar la ejecución del código.

Para ello usados dos clases abstractas: expresiones e instrucciones.

Instrucciones: son las acciones que se realizarán en la etapa de ejecución.

Expresiones: son expresiones de valores que pueden ser asignados a variables.

Clase Instrucción:

Implementaciones de la clase abstracta instrucción, que son las instrucciones del lenguaje Augus.

Cada una hereda de instrucción, pero tiene parámetros diferentes, diseñado para cada instrucción del lenguaje a interpretar.

```
class Instruccion:
    '''Clase abstracta de instrucciones'''

class Print(Instruccion) :

    def __init__(self,  valor) :
        self.valor = valor

class GoTo(Instruccion) :
    '''Clase para la instrucción GoTo Parametro ID'''

    def __init__(self,  id) :
        self.id = id

class Etiqueta(Instruccion) :
    '''Clase instruccion Etiqueta Parametro ID, instrucciones[]'''

    def __init__(self, id, instrucciones=[]) :
        self.id = id
        self.instrucciones=instrucciones


class ExitInstruccion(Instruccion) :
    '''Clase para la instruccion Exit - Sin Parametros '''
    def __init__(self):
```

Clase Expresión:

Implementaciones de la clase abstracta **Expresión,** que son las expresiones reconocidas por Augus.

Que contiene parámetros distintos para cada una, de las cuales son definidas por la gramática.

```
class Expresion:
    '''   Clase para definir Variables '''

class Parametro(Expresion) :
    def __init__(self, expresion) :
        self.expresion=expresion

class Variable(Expresion) :
    ''' Clase que identifica a las variables temporales'''
    def __init__(self, id, tipoVar) :
        self.id=id
        self.tipoVar=tipoVar

class ExpresionValor(Expresion) :

    def __init__(self, val = 0) :
        self.val = val

class AccesoArray(Expresion) :
    def __init__(self,tipoVar,params=[]) :
        self.tipoVar=tipoVar
        self.params=params
```

Inicio de la Definición

En el archivo del analizador importamos las clases de expresiones e instrucciones. Que nos servirán para la construcción del AST.

```
#------------------------------
from expresiones import *
from instrucciones import *
#------------------------------
```

Definición de la Gramática

La forma de declarar producciones en PLY es por medio de una función de Python, seguido de la declaración de una cadena en comilla simples o triples cuando sea de varias líneas.

Así también se puede incrustar reglas semánticas en cada producción que generarán el AST para posterior ejecución.

Fragmento Gramática Ascendente

```
def p_instrucciones_lista(t) :
    'instrucciones    : instrucciones instruccion'
    t[1].append(t[2])
    t[0] = t[1]


def p_instrucciones_instruccion(t) :
    'instrucciones    : instruccion '
    t[0] = [t[1]]

def p_instruccion(t) :
    '''instruccion      : asignacion_instr
                        | asignacion_arr_st
                        | dec_array_instr
                        | print_instr
                        | goto_instr
                        | unset_instr
                        | exit_instr
                        | def_etiqueta_instr
                        | if_instr'''
```

Construcción del AST

La creación del AST requiere que a cada producción se le asocie una regla semántica. En este caso que estamos aplicando el patrón interprete, los nodos serán de un tipo de clase, es decir pueden ser de la clase Instrucción o Expresión.

Asignación de Reglas Semánticas

A cada producción se crea una instancia, en este caso de tipo ExpresionAritmetica, que recibe el valor uno, valor dos y la operación a ejecutar.

```
def p_expresion_aritmetica(t):
    '''expresion_aritmetica : valor MAS valor
                            | valor MENOS valor
                            | valor MULTI valor
                            | valor DIVISION valor
                            | valor RESIDUO valor'''
    if t[2] == '+'  :
        t[0] = ExpresionAritmetica(t[1], t[3], OPERACION_ARITMETICA.MAS)
    elif t[2] == '-':
        t[0] = ExpresionAritmetica(t[1], t[3], OPERACION_ARITMETICA.MENOS)
    elif t[2] == '*':
        t[0] = ExpresionAritmetica(t[1], t[3], OPERACION_ARITMETICA.MULTI)
    elif t[2] == '/':
        t[0] = ExpresionAritmetica(t[1], t[3], OPERACION_ARITMETICA.DIVIDIDO)
    elif t[2] == '%':
        t[0] = ExpresionAritmetica(t[1], t[3], OPERACION_ARITMETICA.RESIDUO)
```

Reglas Semánticas Instrucción

Haciendo uso de las implementaciones de Instrucción se crean reglas semánticas del tipo instrucción que reciban diferentes parámetros.

```
def p_valor_arr_st(t):
    'valor   : tipo_variable lista_parametros'
    t[0] = AccesoArray(t[1],t[2])

def p_valor_read(t) :
    'valor : READ PARIZQ PARDER'
    t[0] = Read()

def p_valor_abs(t) :
    'valor : ABS PARIZQ expresion PARDER'
    t[0] = Absoluto(t[3])

def p_if_instr(t) :
    'if_instr     : IF PARIZQ expresion PARDER goto_instr'
    t[0] =If(t[3], t[5])
```

Explicación: AccesoArray(t[1],t[2])

Se tiene una producción la cual sintetizará los dos atributos y regresará una implementación de la clase instrucción de tipo AccesoArray con dos parámetros.

Producción	Regla Semántica
valor -> tipo_variable lista_parametros	t[0] = AccesoArray(t[1], t[2])

Recuperación de Errores

A cada producción le asignamos una función de la cual contiene la producción de error, la cual descartará la entrada hasta encontrar el carácter siguiente. En este caso, tenemos el carácter de punto y coma con el cual esperamos que se recupere y continúe el análisis y la posterior ejecución.

También declaramos un nuevo error y lo guardamos en una lista para mostrarla en un reporte cuando sea requerido.

```
def p_asignacion_asig_err(t):
    'asignacion_instr : tipo_variable error PTCOMA'
    print("Error sintáctico en '%s'" % (t[2]))
    nErr=ErrorRep('Sintactico','Error de sintaxis en '+str(t[2].value),t[2].lineno)
    lisErr.agregar(nErr)
```

Analizar Cadena

Para analizar una cadena, se llama al parser. Para ello se envía el texto que desea ser analizado, en este caso al hacer uso de dos analizadores léxicos para evitar problemas, se serpararon en dos archivos.

Por lo tanto, le decimos a PLY que el lexer(analizador léxico) que debe ejecutar es el que le enviamos, en este caso será **lexeer=lexer2.**

Nos retornará una lista de instrucciones y expresiones tal como construimos el AST, de esta forma podemos recorrer el AST y ejecutar las instrucciones para finalizar el intérprete.

```
parser = yacc.yacc()
lexer2.lineno=1
par= parser.parse(input,lexer=lexer2)
```

Gramática Descendente

PLY se utiliza con gramáticas ascendentes, sin embargo, podemos implementar el análisis de forma descendente de la siguiente manera:

Uso de la Pila para Gramática Descendente

Al ser PLY un analizador ascendente, para poder generar una gramática descendente hacemos uso de la Pila que nos provee PLY, accediendo a atributos de la pila por medio de un índice negativo. De esta manera, será una forma de heredar atributos.

```
def p_asignacion_valor(t) :
    'asignacion_valor   :   expresion PTCOMA'
    t[0] = Asignacion(t[-2],t[1])

def p_dec_array_instr(t) :
    'asignacion_valor   : ARRAY PARIZQ PARDER PTCOMA'
    t[0] = Array(t[-2])

def p_asignacion_arr_St(t) :
    'asignacion_arr_st   : def_par lista_parametros  IGUAL expresion PTCOMA'
    t[1].extend(t[2])
    t[0] = AsignacionArrSt(t[-1],t[1],t[4])
```

Explicación: t[0]= Asignacion(t[-2],t[1])

```
def p_asignacion_heredada(t) :
    'asignacion_heredada : tipo_variable lista_asignaciones'
    t[0]=t[2]

def p_lista_asignaciones(t) :
    '''lista_asignaciones :  IGUAL asignacion_valor'''
    t[0] = t[2]

def p_asignacion_valor(t) :
    'asignacion_valor   :   expresion PTCOMA'
    t[0] = Asignacion(t[-2],t[1])
```

Producción	Regla Semántica
Asignación_valor-> expresión PTCOMA	t[0]= Asignación(t[-2],t[1])

La clase Asignación recibe como parámetro el id y el valor a asignar.

Para poder recuperar el id en la producción de **asignación_valor,** necesitamos retroceder en la pila, por lo que se obtiene el valor en la posición negativa de la pila.

En el fragmento de la gramática tenemos t[-2] que tiene el valor de **tipo_variable.**

Fragmento de la Gramática Descendente

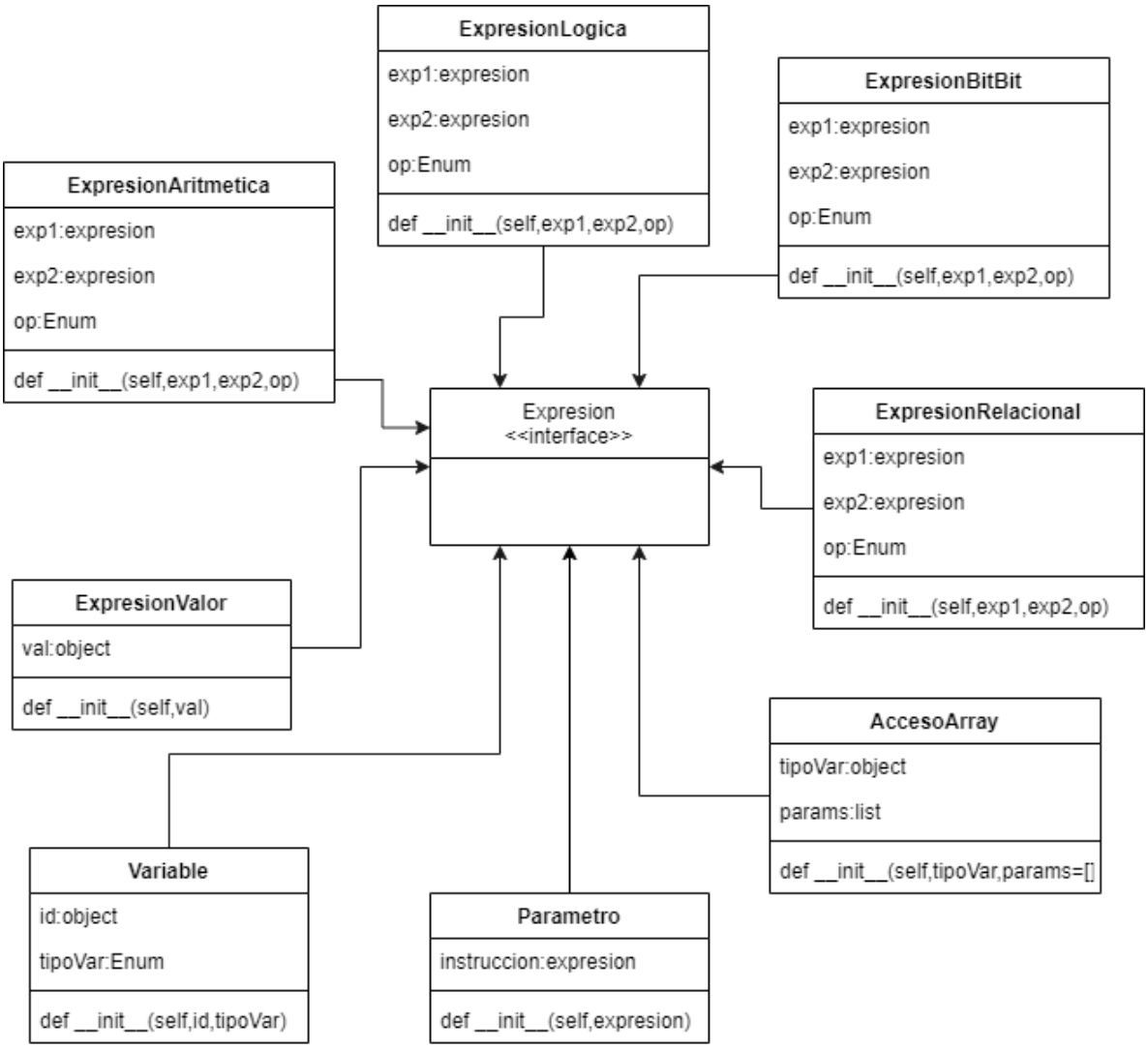
Se elimina la recursividad por la izquierda.

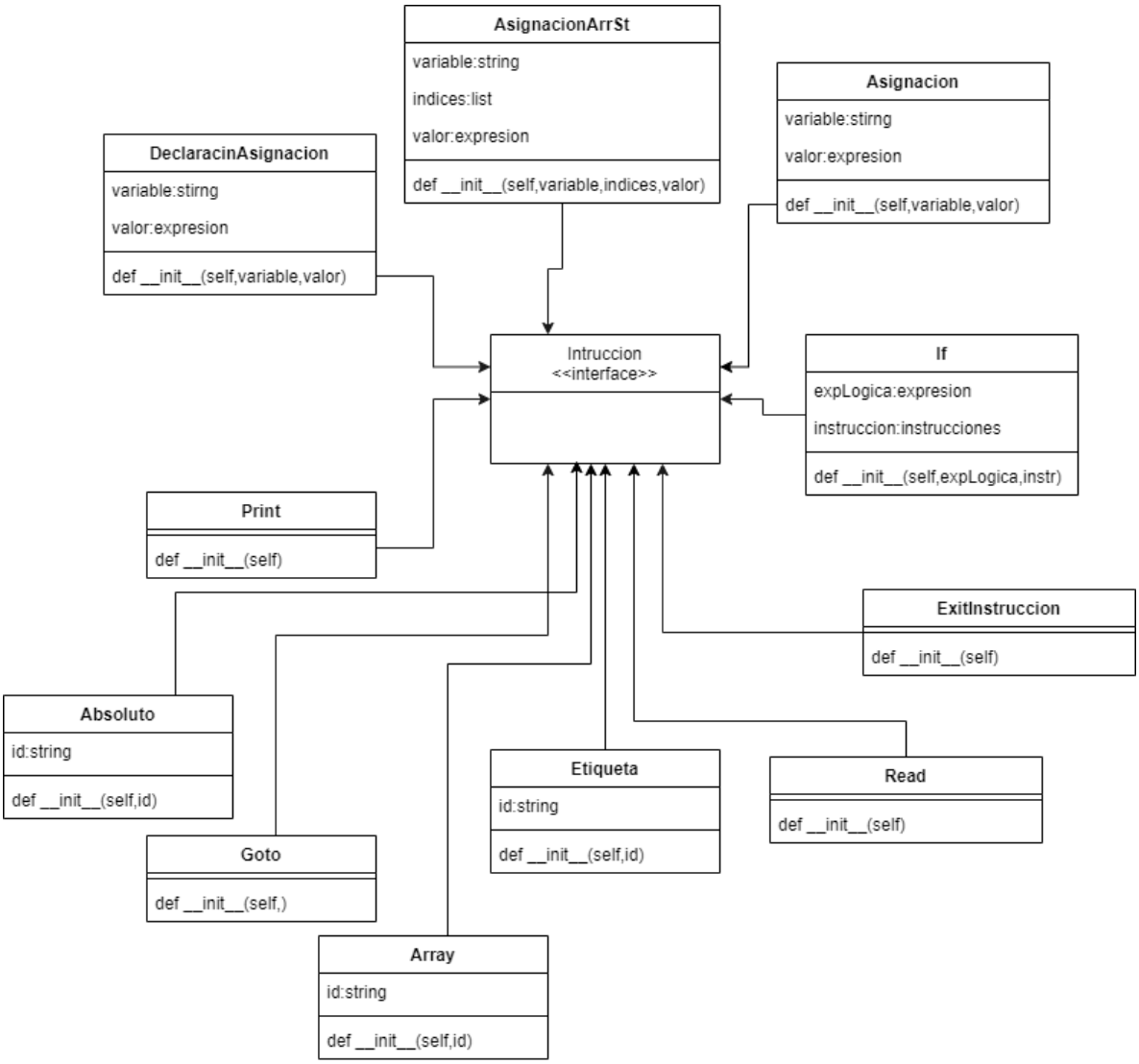
```
def p_init(t) :
    'init : in_instrucciones'
    t[0] = t[1]

def p_in_instrucciones(t) :
    'in_instrucciones : instruccion instrucciones'
    t[1].extend(t[2])
    t[0]=t[1]

def p_instrucciones_lista(t) :
    'instrucciones    : instruccion instrucciones'
    t[1].extend(t[2])
    t[0] = t[1]


def p_instrucciones_empty(t) :
    'instrucciones    :  '
    t[0] = []
```

Se factoriza para eliminar la ambigüedad, tal como debe ser una gramática descendente.

```
def p_expresiones(t) :
    '''expresiones :  MAS valor
                    | MENOS valor
                    | MULTI valor
                    | DIVISION valor
                    | RESIDUO valor
                    | IGUALQUE valor
                    | DISTINTO valor
                    | MAYORIG valor
                    | MENORIG valor
                    | MAYORQUE valor
                    | MENORQUE valor
                    | AND valor
                    | OR valor
                    | XOR valor
                    | ANDBB valor
                    | ORBB valor
                    | XORBB valor
                    | SHIFTIZQ valor
                    | SHIFTDER valor
                    | empty
    '''
```

ANEXOS


ExpresionLogica
exp1:expresion
exp2:expresion
op:Enum
def __init__(self,exp1,exp2,op)
ExpresionBitBit
exp1:expresion
exp2:expresion
op:Enum
def __init__(self,exp1,exp2,op)
ExpresionAritmetica
exp1:expresion
exp2:expresion
op:Enum
def __init__(self,exp1,exp2,op)
Expresion
<<interface>>
ExpresionRelacional
exp1:expresion
exp2:expresion
op:Enum
def __init__(self,exp1,exp2,op)
ExpresionValor
val:object
def __init__(self,val)
AccesoArray
tipoVar:object
params:list
def __init__(self,tipoVar,params=[]
Variable
id:object
tipoVar:Enum
def __init__(self,id,tipoVar)
Parametro
instruccion:expresion
def __init__(self,expresion)


AsignacionArrSt
variable:string
indices:list
valor:expresion
def __init__(self,variable,indices,valor)
Asignacion
variable:stirng
valor:expresion
def __init__(self,variable,valor)
DeclaracinAsignacion
variable:stirng
valor:expresion
def __init__(self,variable,valor)
Intruccion
<<interface>>
If
expLogica:expresion
instruccion:instrucciones
def __init__(self,expLogica,instr)
Print
def __init__(self)
ExitInstruccion
def __init__(self)
Absoluto
id:string
def __init__(self,id)
Etiqueta
id:string
def __init__(self,id)
Read
def __init__(self)
Goto
def __init__(self,)
Array
id:string
def __init__(self,id)