Manual Tecnico AUGUS

ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 2

JAVIER GOLON
201700473

DESCRIPCION LENGUAJE AUGUS

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

Es un lenguaje débilmente tipado, sin embargo, si se reconocen cuatro tipos de datos no explícitos: entero, punto flotante, cadena de caracteres y arreglo. Para manejar el flujo de control se proporciona la declaración de etiquetas, sin tener palabras reservadas para ese uso. Es decir, no hay ciclos for, while, ni do-while.

RESUMEN IMPLEMANTACION

Se utilizaron conceptos de la Traducción dirigida por la sintaxis para construir un intérprete para el lenguaje para el cual se hizo uso de la herramienta PLY(explicación más adelante) más diversas tecnologías que se explicaran de forma más detallada en la sección de Herramientas, así mismo se implementó para este caso un análisis ASC y un DESC simulado por el manejo de pila permitido por PLY, también se explicara más adelante.

Herramientas

Python

Python es un lenguaje de programación interpretado cuya filosofía hace hincapié en la legibilidad de su código.2 Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, dinámico y multiplataforma.

Es administrado por la Python Software Foundation. Posee una licencia de código abierto, denominada Python Software Foundation License,3 que es compatible con la Licencia pública general de GNU a partir de la versión 2.1.1, e incompatible en ciertas versiones anteriore

Version Utlilizada " Python 3.8.3 "
https://www.python.org/downloads/windows/

PLY

es una herramienta de análisis escrita exclusivamente en Python . Es, en esencia, una reimplementación de Lex y Yacc originalmente en lenguaje C. Fue escrito por David M. Beazley . PLY utiliza la misma técnica de análisis LALR que Lex y Yacc. También cuenta con amplias instalaciones de depuración y reporte de errores.

Version Utilizada 3.11
https://www.dabeaz.com/ply/

Graphviz

Graphviz es un conjunto de herramientas de software para el diseño de diagramas definido en el lenguaje descriptivo DOT. Fue desarrollado por AT&T Labs y liberado como software libre con licencie tipo Eclipse

Version Utilizada 2.44
https://graphviz.org/download/

Entorno de desarrollo

Dell latitud 3576 Intel i7 octava generacion, 12 GB de ram, 1tb almacenamient HDD
Sistema operativo x64 Windows 10 home edition

Librerias Utilizadas

Tkinter (Integrada)
Tkdocviewer (2.0.1) https://pypi.org/project/tkDocViewer/
Pillow (7.1.2) https://pillow.readthedocs.io/en/stable/

Desarrollo:

Para interpretar el lenguaje AUGUS se utilizo como lenguaje base python, en donde se implementaron las herramientas de analisis lexico y sintactico (PLY)

Analizadores:

Gramatica.py
GramaticaDESCpy

El archivo gramatica contiene la estructura para un analisis ascendente del lenguaje AUGUS, GramaticaDESC contiene la misma gramatica pero factorizada y sin recursividad por la izquierda para poder cumplir con las reglas de una gramatica para un analizador descendente

Dentro de ambas se encuentran la declaracion de terminales y no terminales

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

```
tokens  = [
    'PTCOMA',
    'CORCHETEIZQ',
    'CORCHETEDER',
    'PARIZQ',
    'PARDER',
    'IGUAL',
    'MAS',
    'MENOS',
    'POR',
    'DIVIDIDO',
    'RESIDUO',
    'ANDBIT',
    'ANDLOGICA',
    'MENOR',
    'MAYOR',
    'MAYORIGUAL',
    'MENORIGUAL',
    'IGUALIGUAL',
    'NOIGUAL',
    'XORBIT',
    'ORBIT',
    'ORLOGICA',
    'SHIFTIZQ',
    'SHIFTDER',
```

```
t_PTCOMA    = r';'
t_PARIZQ    = r'\('
t_PARDER    = r'\)'
t_CORCHETEIZQ = r'\['
t_CORCHETEDER = r'\]'
t_IGUAL     = r'='
t_MAS       = r'\+'
t_MENOS     = r'-'
t_POR       = r'\*'
t_DIVIDIDO  = r'/'
t_RESIDUO   = r'%'
t_ANDBIT    = r'&'
t_ANDLOGICA =r'&&'
t_MENOR     = r'<'
t_MAYOR     = r'>'
t_MAYORIGUAL= r'>='
t_MENORIGUAL= r'<='
t_IGUALIGUAL  = r'=='
t_NOIGUAL = r'!='
t_XORBIT = r'\^'
t_ORBIT= r'\|'
t_ORLOGICA= r'\|\|'
t_SHIFTIZQ=r'<<'
t_SHIFTDER=r'>>'
t_NOTBIT=r'~'
t_NOTLOGICA='!'
```

Para las expresiones como identificadores, numeros y variables del lenguaje AUGUS se hizo uso de las expresiones regulares

```
def t_DECIMAL(t):
    r'\d+\.\d+'
    try:
        t.value = float(t.value)
    except ValueError:
        print("Float value too large %d", t.value)
        t.value = 0
    return t

def t_ENTERO(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print("Integer value too large %d", t.value)
        t.value = 0
    return t

def t_TEMPORALES(t):
    r"\$t(\d+)"
    t.type=reservadas.get(t.value.lower(),'TEMPORALES')
    return t
```

Asi mismo el manejo de caracteres o secuencias no reconocidas como comentarios o espacios en blanco, tambien el manejo de los errores para su identificacion posterior

```
# Caracteres ignorados
t_ignore = " \t"

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

def find_column(input, token):
    line_start = input.rfind('\n', 0, token.lexpos) + 1
    return (token.lexpos - line_start) + 1

def t_error(t):

    print("Illegal character '%s'" % t.value[0])
    mistake=LErrores.Error('Lexical Error',str(t.value[0]),'Ilegal Character',t.lexer.lineno,find_column(
    ListaE.AddError(mistake)
    t.lexer.skip(1)

lexer = lex.lex()
```

Gramaticas:

Para el interprete de AUGUS se utilizaron dos formas de analisis, ascendente y descendente siendo el descente una forma simulada por medio de la manipulacion de pila que permite PLY, las gramaticas respectivas se encuentran adjuntas a este documento pero las diferencias significativas con las siguientes

```
# ========================================================================
# ======================================= DESC ===========================
# ========================================================================

def p_init(t):
    'init  :    MAIN DOSPUNTOS listainstrucciones'
    t[0]=t[3]
def p_lista_instrucciones(t):
    'listainstrucciones :   simpleinstrucciones listainstruccionesp'
    t[0]=t[2]

def p_lista_instrucciones_prima(t):
    'listainstruccionesp : simpleinstrucciones listainstruccionesp'
    t[0] = t[2]
    t[0].insert(0,t[-1])
def p_lista_instrucciones_epsilon(t):
    'listainstruccionesp    :   '
    t[0] = [t[-1]]

# ========================================================================
# ======================================= ASC ============================
# ========================================================================

def p_init(t):
    'init  :    MAIN DOSPUNTOS listainstrucciones'
    t[0]=t[3]
def p_lista_instrucciones(t):
    'listainstrucciones :   listainstrucciones simpleinstrucciones'
    t[1].append(t[2])
    t[0]=t[1]
def p_lista_instrucciones_simple(t):
    'listainstrucciones : simpleinstrucciones'
    t[0]=[t[1]]
```

La forma asc consta de producciones recursivas por la izquierda por lo que crear el arreglo de instrucciones se hace agregando al final de la lista mientras que en el desc de forma natural genera un arbol invertido por lo que para poder trabajar con el misma estructura del proyecto se insertaron los nuevos nodos el inicio para asi tener al final del analisis la misma estructura sin importar el orden de evaluacion

Patron de diseño

El interpreter es un patrón de diseño que, dado un lenguaje, define una representación para su gramática junto con un intérprete del lenguaje.

Para el analisis y ejecucion de la entrada se hizo uso del patron interprete por medio de clases y herencias

El objetivo principal de nuestro analizador sintáctico es validar que la entrada sea válida y, si lo es, construir el AST. Para lograr esto hacemos uso de la programación orientada a objetos. Específicamente haremos uso del polimorfismo para la construcción de nuestro árbol. Las clases utilizadas para construir las diferentes instrucciones que componen nuestro AST, están definidas en el archivo instrucciones.py.

Primero definimos una clase abstracta Instruccion, esto nos permitirá abstraer las Instrucciones que soporta nuestro lenguaje:

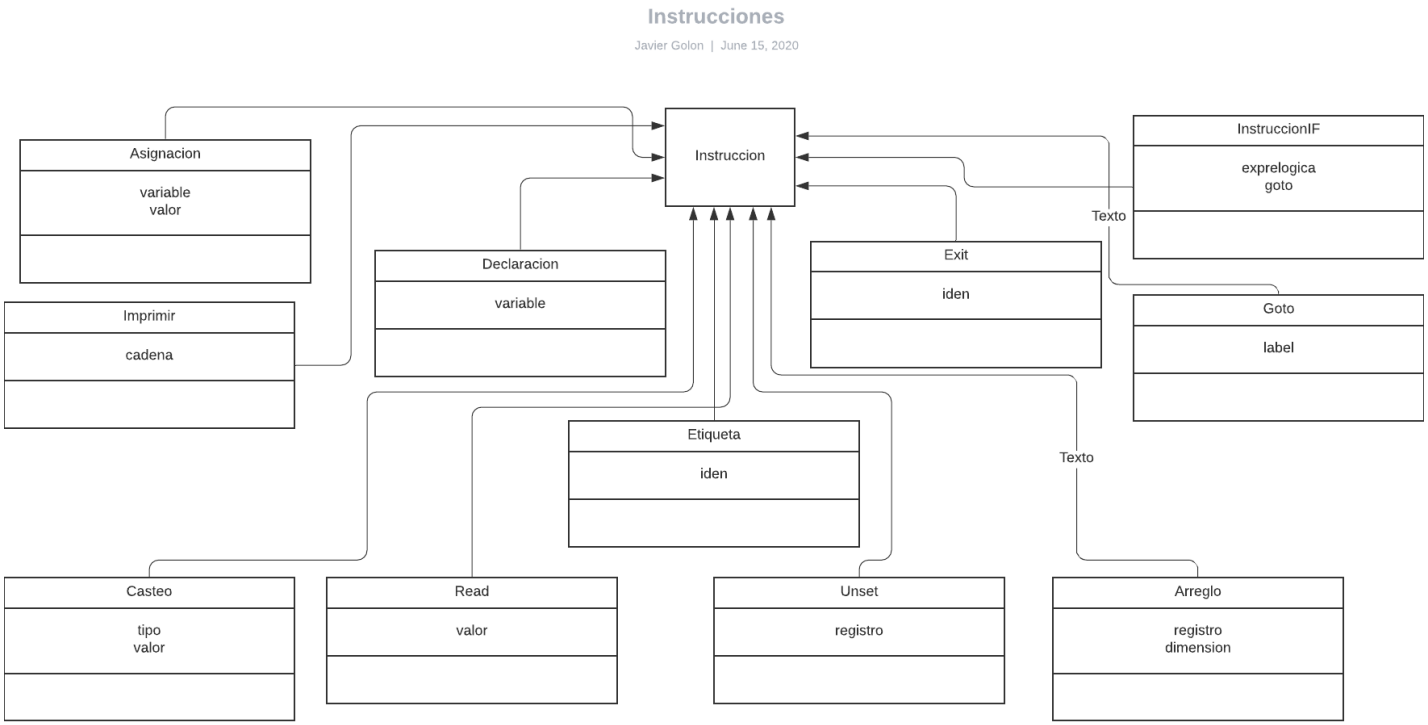
```
class Instruccion:
    ''' Representa la clase padre de las instrucciones'''
#====================== CLASES QUE HEREDAN DE LA CLASE INSTRUCCION ==============
```

Seguidamente, definimos una clase concreta para cada una de las formas posibles que puede tomar Instruccion:

```
class declaracion(Instruccion):
    def __init__(self,variable):
        self.variable=variable

class asignacion(Instruccion):
    def __init__(self,variable,valor):
        self.variable=variable
        self.valor=valor

class instruccionIf(Instruccion):
    def __init__(self,exprelogica,goto):
        self.exprelogica=exprelogica
        self.goto=goto

class Exit(Instruccion):
    def __init__(self,iden):
        self.iden=iden

class Imprimir(Instruccion):
    def __init__(self,cadena):
        self.cadena=cadena

class Etiqueta(Instruccion):
    def __init__(self,iden):
        self.iden=iden
```

Por ejemplo, para la clase Imprimir vemos que extiende de Instruccion y que su única propiedad es la cadena que se va imprimir. Esta propiedad, cadena, es de tipo ExpresionCadena como veremos más adelante.

La siguiente imagen ilustra la estructura



Clases para Expresiones

De la misma manera que manejamos las instrucciones manejaremos las expresiones. Definimos 3 clases abstractas que representan los 5 tipos de expresiones soportadas por nuestro lenguaje: ExpresionesNumerica, ExpresionesComillas, ExpresionesLógicas, ExpresionBit, Expresion_Relacional, Expresion dimension todas ellas definidas dentro del archivo expresiones.py.

```python
class ExpresionNumerica:
    ''' clase maestra que representa una expresion numerica'''
class ExpresionLogica:
    ''' clase que representa una expresion logica'''
class ExpresionBit:
    ''' clase que representa una expresion logica'''
class ExpresionRelacional:
    ''' clase que representa una expresion logica'''
```

También haremos uso de enumeraciones para definir constantes de nuestras operaciones, esto es altamente recomendado para evitar bugs durante el desarrollo.

```python
from enum import Enum

class OPERACION_ARITMETICA(Enum):
    MAS=1
    MENOS=2
    DIVIDIDO=3
    POR=4
    RESIDUO=5
    ABSOLUTO=6
class OPERACION_LOGICA(Enum):
    NOT=1
    AND=2
    OR=3
    XOR=4

class OPERACION_BIT(Enum):
    NOT=1
    AND=2
    OR=3
    XOR=4
    SHIFTIZQ=5
    SHIFTDER=6

class OPEREACION_RELACIONAL(Enum):
    IGUAL=1
    DIFERENTE=2
    MAYORIGUAL=3
    MENORIGUAL=4
    MAYOR=5
    MENOR=6
```

Las diferentes formas que las clases pueden tomar forma son

```
class ExpresionBinariaRelacional(ExpresionRelacional):
    def __init__(self,exp1,exp2,operador):
        self.exp1=exp1
        self.exp2=exp2
        self.operador=operador

class ExpresionBinariaBit(ExpresionBit):
    def __init__(self,exp1,exp2,operador):
        self.exp1=exp1
        self.exp2=exp2
        self.operador=operador

class ExpresionBinarioLogica(ExpresionLogica):
    def __init__(self,exp1,exp2,operador):
        self.exp1=exp1
        self.exp2=exp2
        self.operador=operador

class ExpresionBinariaAritmetica(ExpresionNumerica):
    def __init__(self,exp1,exp2,operador):
        self.exp1=exp1
        self.exp2=exp2
        self.operador=operador

class ExpresionMonoLogica(ExpresionLogica):
    def __init__(self,exp,operador):
        self.exp=exp
        self.operador=operador
```

Todas siguen la misma estructura , reciben tres parametros, excepto las expresiones de un solo operando
Exp1 = operando de la izquierda
Exp2 = operando de la derecha
Operador = operador de la expresion (detalladas en el lexico)

La tabla de símbolos

La tabla de símbolos es la que permite el almacenamiento y recuperación de los valores de las variables. Para su implementación hacemos uso de una clase, ya que necesitaremos más de una instancia de tabla de símbolos. Cada ámbito tiene acceso únicamente a su propia tabla de símbolos y a la de los niveles superiores, la definición de esta clase puede encontrarse en el archivo TablaSimbolos.py.

Definimos las constantes para los tipos de datos

```python
from enum import Enum

class TIPO_DATO(Enum) :
    NUMERO = 1
    CADENA=2
    FLOAT=3
    METODO=4
    FUNCION=5
    ETIQUETA=6
    ARRAY=7
    STRUCT=8
```

Definimos una clase para los Símbolos.

```python
class Simbolo() :
    'Esta clase representa un simbolo dentro de nuestra tabla de simbolos'

    def __init__(self,tipo, valor) :
        self.tipo = tipo
        self.valor = valor # para etiqueta valor sera mi posicion
```

La clase TablaDeSimbolos define la estructura de una tabla de símbolos y sus funciones para agregar, modificar y obtener símbolos.

```python
class TablaDeSimbolos() :
    'Esta clase representa la tabla de simbolos'

    def __init__(self, simbolos = {}) :
        self.simbolos = simbolos

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

    def obtener(self, id) :
        if not id in self.simbolos :
            #print('Error: variable ', id, ' no definida.')
            return None # ya se puede asignar o crear

        return self.simbolos[id]

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

    def EliminarSimbolo(self,id):
        self.simbolos.pop(id)

    def ObtenerTabla(self):
        return self.simbolos
```

Construcción del Intérprete

La definición del Intérprete se encuentra en el archivo Ejecucion.py

Para iniciar con la implementación, primero importamos nuestra gramática, las constantes y clases de nuestro AST y la Tabla de Símbolos.

```python
import Gramatica as g # gramatica asc
import GramaticaDESC as FormaDesc
import TablaSimbolos as TS
from operator import xor
from Metodos_Complementarios import *
from Expresiones import *
from Instrucciones import *
from GraficarAST import Graficadora
import tkinter as tk
from tkinter import simpledialog
import Globales
import re
```

La función principal del intérprete es de reconocer cada instrucción y ejecutarla, para esto es necesario recorrer el AST; es por ello que se ha definido la función

```
def Recorrer_Instrucciones(instrucciones,ts):
    accion_LlenarTsEtiquetas(instrucciones,ts) # mando a llenar la ts con todas mis etiquetas
    largo = len(instrucciones)
    i = 0
    l = resetable_range(largo)
    for i in l:
        if isinstance(instrucciones[i],Imprimir): accion_imprimir(instrucciones[i],ts)
        elif isinstance(instrucciones[i],asignacion): accion_asignar(instrucciones[i],ts)
        elif isinstance(instrucciones[i],declaracion): accion_declaracion(instrucciones[i],ts)
        elif isinstance(instrucciones[i],Unset): accion_unset(instrucciones[i].registro,ts)
        elif isinstance(instrucciones[i],instruccionIf):
            estado = resolver_Expresion(instrucciones[i].exprelogica,ts)
            if estado == 0 :
                ''' sigue el flujo del programa '''
            elif estado == 1:
                newindex = ts.obtener(instrucciones[i].goto).valor # la etiqueta fijo esta en ts
                i= newindex # salto a la posicion donde viene la etiqueta para hacer los saltos
                l.reset(i)
                continue
        elif isinstance(instrucciones[i],Goto):
            newindex = ts.obtener(instrucciones[i].label).valor #la etiqueta fijo esta en ts
            i=newindex
            l.reset(i)
            continue
        elif isinstance(instrucciones[i],Exit):
            i = largo # termina la ejecucion de mi codigo
            l.reset(i)
```

Cabe resaltar que ya sea para el analisis ascendente como el descendente la implementacion y la ejecucion es la misma por lo cual el arbol que genera(AST) es el mismo. Para poder difereciarlos se genera un reporte gramatical con las formas y acceso al momento del anlisis sintactico incluyendo las reglas gramaticales respectivas. Tambien se resalta que el cambio de una forma a otra no varia mas que en las listas de acceso

Interfaz

La interfaz se encuntra en el archivo interfaz.py la cual al momento de completar el proyecto se convierte en el archivo principal de ejecucion, de el se hace el llamado a las demas clases para poder funcionar

```
import tkinter
import os
from tkinter import ttk
from tkinter.messagebox import *
from tkinter.filedialog import *
from tkinter.simpledialog import *
from tkdocviewer import *
import re
# Importacion de la clase ejecucion
import Ejecucion as analisador
import Globales
Globales.initialized()
```

La herramienta utilizada es tikinter, librería nativa de python