

Universidad de San Carlos de Guatemala  
Facultad de Ingeniería  
Ciencias y Sistemas  
Organización de Lenguajes y Compiladores 1  
Sección: N  
Nombre: Joel Obdulio Xicará Ríos  
Carnet: 201403975

---

***Manual Técnico – JPR***

---

*Guatemala, 04 de julio de 2021*

# Funciones y Procedimientos de JPR

Link del Repositorio: <https://github.com/JoelX09/OLC1-ProyectoVJ-201403975.git>

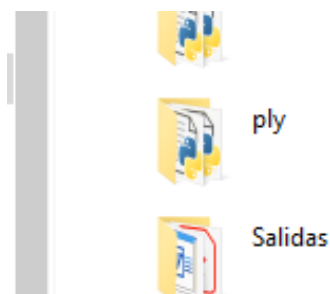
## Lenguaje de programación utilizado:

Lenguaje utilizado para el desarrollo del programa: **Python**. Se utilizó la versión: **3.9.5 64-bit**

## Herramientas para el análisis del archivo de entrada:

Se utilizó ply-3.11 para el análisis léxico y sintáctico de la cadena de entrada. Se puede descargar del siguiente enlace: <https://www.dabeaz.com/ply/>

Descomprimir el .rar que se descarga, y mover la carpeta ply al directorio donde se encuentra nuestro proyecto.



Ply es un analizador LALR de tipo ascendente.

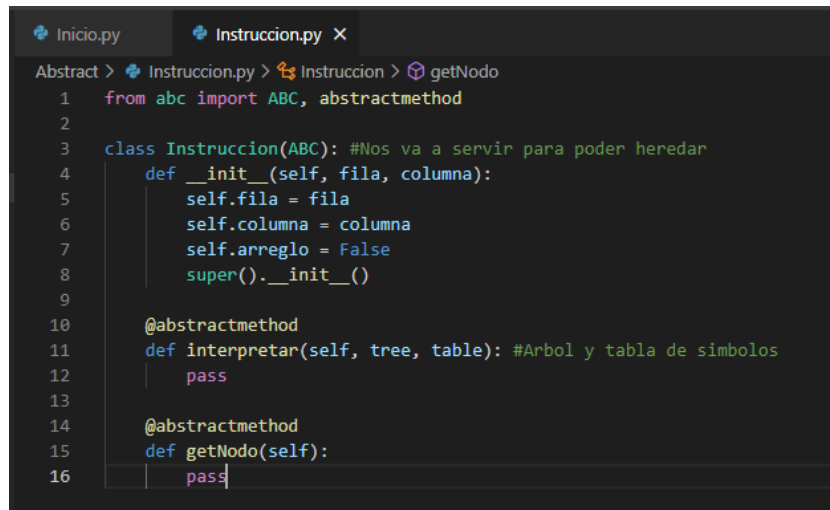
## Librería utilizada para la creación de la interfaz:

Se utilizó la librería **tkinter** para el desarrollo de la interfaz gráfica, esta librería ya viene incluida al instalar Python por lo que no es necesario realizar algún paso extra más que importarla en nuestro código, se puede realizar de tres formas distintas:

```
Inicio.py • Excepcion.py
Inicio.py > ...
1 import tkinter as tk
2
3 from tkinter import *
4
5 from tkinter import Button, PhotoImage, Canvas
6 from tkinter import BOTH, INSERT, END, LEFT
```

## Patron Interprete

Para el desarrollo de la lógica del funcionamiento de nuestro programa se utilizó el patron interprete por medio del cual creamos una clase abstracta llamada **Instrucción**, de la cual se va a heredar un método llamada **interpretar** y que vas a estar presente en todas nuestras instrucciones propias del sistema que se ejecutaran. Este método permite manejar cualquier tipo de dato u objeto pues con este método se obtiene de que tipo es. También se tiene un método abstracto llamado **getNode** el cual se utilizara para la creación del AST



```
Inicio.py Instruccion.py X
Abstract > Instruccion.py > Instruccion > getNode
1 from abc import ABC, abstractmethod
2
3 class Instruccion(ABC): #Nos va a servir para poder heredar
4     def __init__(self, fila, columna):
5         self.fila = fila
6         self.columna = columna
7         self.arreglo = False
8         super().__init__()
9
10    @abstractmethod
11    def interpretar(self, tree, table): #Arbol y tabla de simbolos
12        pass
13
14    @abstractmethod
15    def getNode(self):
16        pass
```

Para la implementación del método del árbol se hicieron uso de cinco clases fundamentales:

### Clase Árbol:

La clase abstracta del patron intérprete emplea un objeto llamado tree, el cual es una instancia de una clase llamada Árbol y que contiene todos los atributos necesarios en la ejecución del programa. Para el acceso y modificación de cada atributo de la clase Árbol se cuentan con sus set and getters.

```

Inicio.py  Arbol.py x
5 > Arbol.py > Arbol
1 class Arbol:
2     def __init__(self, instrucciones):
3         self.instrucciones = instrucciones #Clase abstracta que puede ser cualquier cosa
4         self.funciones = [] #Pull de funciones
5         self.excepciones = []
6         self.consola = ""
7         self.TSglobal = None
8         self.ventana = None
9         self.salida = None
10        self.textoRead = "Read"
11        self.dot = ""
12        self.contador = 0
13        self.simbolos = {}
14        self.entorno = "Global"
15
16    def getSimbolos(self):
17        return self.simbolos
18
19    def addSimbolo(self, simbolo):
20        self.simbolos[simbolo.identificador.lower()] = simbolo
21
22    def addSimboloF(self, simbolo):
23        self.simbolos[str(simbolo.identificador.lower()) + "##Funcion"] = simbolo
24
25    def updateSimbolo(self, identificador, tipo, valor):
26        self.simbolos[identificador.lower()].setTipo(tipo)
27        self.simbolos[identificador.lower()].setValor(valor)
28
29    def updateSimboloF(self, identificador, tipo, valor):
30        self.simbolos[str(identificador.lower()) + "##Funcion"].setTipo(tipo)
31        self.simbolos[str(identificador.lower()) + "##Funcion"].setValor(valor)
32
33    def getInstrucciones(self):

```

## Clase Excepción:

Para el manejo de los errores se crean objetos de tipo Excepción. Una excepción contiene los datos necesarios para poder corregir los errores que se detecten en el análisis del archivo de entrada.

Se cuenta con un método llamado toString el cual retorna una cadena con los datos del error registrado.

```

Inicio.py  Excepcion.py x
TS > Excepcion.py > Excepcion > __init__
1 class Excepcion: #Guarda errores, parecida a la de simbolos
2     def __init__(self, tipo, descripcion, fila, columna):
3         self.tipo = tipo
4         self.descripcion = descripcion
5         self.fila = fila
6         self.columna = columna
7
8     def toString(self):
9         return self.tipo + " - " + self.descripcion + " [" + str(self.fila) + "," + str(self.columna) + "]"

```

## Clase Símbolo:

Esta clase contiene la estructura de los símbolos que se van a ir creando dentro del lenguaje que se está declarando. Estructura principalmente de las variables. Para la obtención o modificación de un atributo en específico de un Símbolo se cuentan con los setters y getters de cada atributo.

```
Simbolo.py X
TS > Simbolo.py > Simbolo
1 class Simbolo:
2     def __init__(self, identificador, tipo, arreglo, fila, columna, valor): #Constructor
3         self.id = identificador
4         self.tipo = tipo
5         self.arreglo = arreglo
6         self.fila = fila
7         self.columna = columna
8         self.valor = valor
9
10    def getId(self):
11        return self.id
12
13    def setId(self, id):
```

## Clase TablaSimbolos:

Esta clase contiene la tabla en la que se irán almacenando los símbolos que se van a ir creando dentro del lenguaje que se está declarando. Cada entorno o ámbito que se pueda encontrar en el lenguaje tendrá una tabla de símbolos. Se tiene el método setTabla() para agregar un nuevo símbolo, getTabla() para obtener un símbolo, y actualizarTabla() para actualizar un símbolo.

```
Simbolo.py TablaSimbolos.py X
TS > TablaSimbolos.py > TablaSimbolos
1 from TS.Excepcion import Excepcion
2 from TS.Tipo import TIPO
3
4 class TablaSimbolos:
5     def __init__(self, anterior = None):
6         self.tabla = {} #Diccionario vacio, es como una tabla hash
7         self.anterior = anterior
8
9     def setTabla(self, simbolo): #Agregar una variable/Simbolo
10        if simbolo.id.lower() in self.tabla:
11            return Excepcion("Semantico", "Variable " + simbolo.id + " ya existe", simbolo.fila, simbolo.col)
12        else:
13            self.tabla[simbolo.id.lower()] = simbolo
14            return None #Se agrego correctamente
15
16    def getTabla(self, id): #Obtener una variable/Simbolo
17        tablaActual = self
18        while tablaActual != None:
```

## Clase Tipo:

En esta clase se definen los tipos más básicos que de dato y operador que se pueden manejar dentro del lenguaje, se utiliza la librería **enum** para asignarles un índice y tener un orden:

### Tipo de Dato:

```
class TIPO(Enum):  
    ENTERO = 1  
    DECIMAL = 2  
    BOOLEANO = 3  
    CHARACTER = 4  
    CADENA = 5  
    NULO = 6  
    ARREGLO = 7
```

### Operador Aritmético:

```
class OperadorAritmetico(Enum):  
    MAS = 1  
    MENOS = 2  
    POR = 3  
    DIV = 4  
    POT = 5  
    MOD = 6  
    UMENOS = 7
```

### Operador Relacional:

```
class OperadorRelacional(Enum):  
    MENORQUE = 1  
    MAYORQUE = 2  
    MENORIGUAL = 3  
    MAYORIGUAL = 4  
    IGUALIGUAL = 5  
    DIFERENTE = 6
```

### Operador Lógico:

```
class OperadorLogico(Enum):  
    NOT = 1  
    AND = 2  
    OR = 3
```

### Método para analizar la entrada:

**parse()** Es el método principal por medio del cual se ejecuta el analizador, asignamos las variables de los elementos que serán necesarios y en return hace la llamada al parse perteneciente a Ply.

```
def parse(inp) :
    global errores
    global lexer
    global parser
    errores = []
    lexer = lex.lex()
    parser = yacc.yacc()
    global input
    input = inp
    return parser.parse(inp)
```

### Funciones Nativas:

Las funciones nativas, son funciones propias del sistema que existen por defecto, estas se crean al iniciar el análisis y se ingresan al árbol.

```
def crearNativas(ast):          # CREACION Y DECLARACION DE LAS FUNCIONES NATIVAS
    nombre = "toupper"
    parametros = [{'tipo':TIPO.CADENA,'identificador':'toupper##Param1','arreglo':False}]
    instrucciones = []
    toUpper = ToUpper(nombre, parametros, instrucciones, -1, -1)
    ast.addFuncion(toUpper)      # GUARDAR LA FUNCION EN "MEMORIA" (EN EL ARBOL)
```

### Llamada al analizador en la interfaz:

Con el botón interpretar hacemos el llamado al método parse explicado anteriormente y se obtiene la lista de instrucciones que existen en el lenguaje para poder interpretarlas y darle al programa el funcionamiento que supone debe tener. Aquí se crean y meten las funciones nativas haciendo el llamado al método anteriormente detallado.

```
instrucciones = parse(entrada)
ast = Arbol(instrucciones)
ast.ventana = root
ast.salida = consola
TSGlobal = TablaSimbolos()
ast.setTSGlobal(TSGlobal)
crearNativas(ast)
```