

Universidad de San Carlos de Guatemala
Facultad de Ingenieria
Ciencias y Sistemas
Organización de Lenguajes y Compiladores 1
Laboratorio Seccion N
Tutor Academico: José Puac

Manual Técnico

JPR

Nombre: Jorge Isaac Xicol Vicente
Carnet: 201807316

Introducción

Este proyecto consiste en realizar un interprete con un lenguaje especificado. El proyecto es nombrado JPR y en este documento se describiran las especificaciones del lenguaje programado, juntamente con los metodos e ideas realizadas.

Entorno de Trabajo

Editor:

El editor sera de ayuda al usuario para poder realizar el analisis, reportes y poder abrir archivos por medio de un entorno agradable de usuario. La librería usada para realizar este editor fue **Tkinter de python**. Algunas de las librerias usadas para este proyecto relacionadas al entorno son las siguientes:

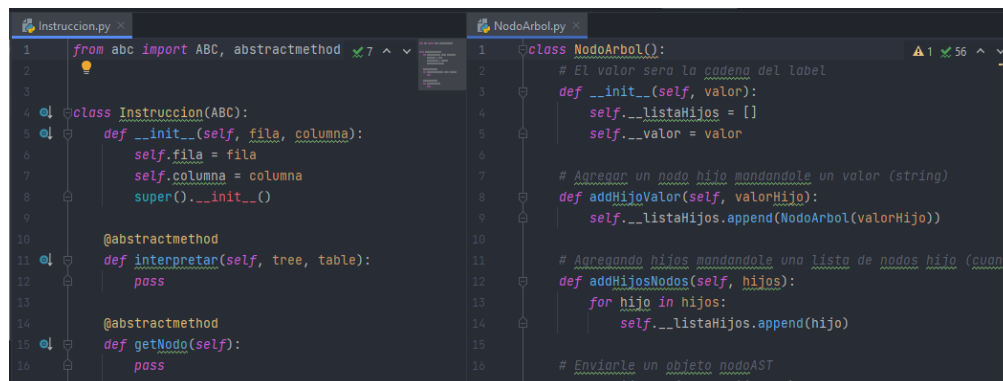
- Filedialog
- Srolledtext
- Simpliedialog – askstring
- Webbrowser – opennewtab

Patrón Interprete

Para poder cosntruir el proyecto, se hizo uso del patron interprete que es usado para definir patrones por medio de tecnicas como programacion ortientada a objetos, herencia, polimorfismo y recursividad.

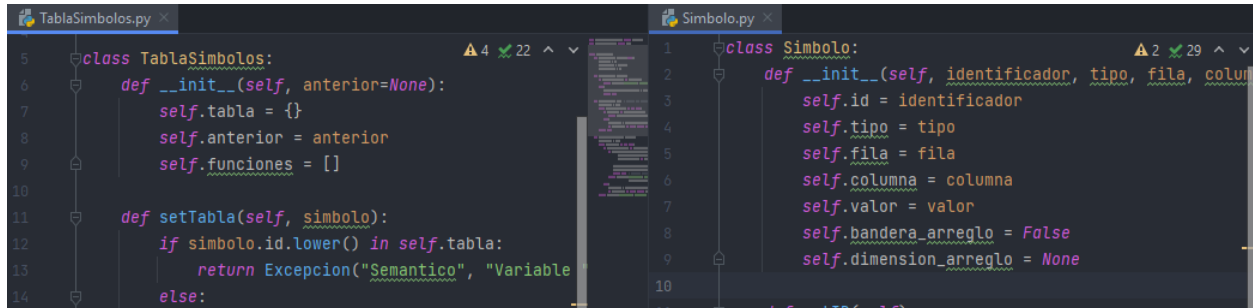
Los aspectos importantes usados en este proyecto son explicados a continuacion.

- Utilizacion de clase abstractas: Se usaron las clases abstractas para poder implmentar polimorfismo en las distintas clases, ya que cada metodo tendria en este caso un metodo “interpretar()” y “getNodo()” con funcionalidades totalmente distintas.



```
1 from abc import ABC, abstractmethod
2
3
4 class Instruccion(ABC):
5     def __init__(self, fila, columna):
6         self.fila = fila
7         self.columna = columna
8         super().__init__()
9
10    @abstractmethod
11    def interpretar(self, tree, table):
12        pass
13
14    @abstractmethod
15    def getNodo(self):
16        pass
17
18
19 class NodoArbol():
20     # El valor sera la cadena del label
21     def __init__(self, valor):
22         self.__listaHijos = []
23         self.__valor = valor
24
25     # Agregar un nodo hijo mandandole un valor (string)
26     def addHijoValor(self, valorHijo):
27         self.__listaHijos.append(NodoArbol(valorHijo))
28
29     # Agregando hijos mandandole una lista de nodos hijo (cuando
30     def addHijosNodos(self, hijos):
31         for hijo in hijos:
32             self.__listaHijos.append(hijo)
33
34     # Enviarle un objeto nodoAST
35     def addHijoNodo(self, hijoNodo):
```

- Ambitos: Para la realización de ambitos, se trabajo con un objeto Tabla Simbolos que tendria una apuntador anterior, que puntaria a su ambito anterior creando una lista enlazada simple de tablas de simbolos. En las tablas se almacenan tambien los Simbolos, que serian equivalentes a las variables, funciones, metodos declarados en el programa.



```

TablaSimbolos.py
5 class TablaSimbolos:
6     def __init__(self, anterior=None):
7         self.tabla = {}
8         self.anterior = anterior
9         self.funciones = []
10
11     def setTabla(self, simbolo):
12         if simbolo.id.lower() in self.tabla:
13             return Excepcion("Semantico", "Variable
14         else:

```

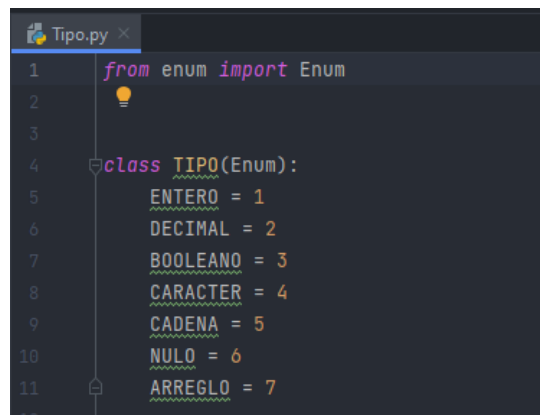
```

Simbolo.py
1 class Simbolo:
2     def __init__(self, identificador, tipo, fila, colum
3         self.id = identificador
4         self.tipo = tipo
5         self.fila = fila
6         self.columna = columna
7         self.valor = valor
8         self.bandera_arreglo = False
9         self.dimension_arreglo = None
10

```

Al momento de buscar una variable, se tendria que recorrer todas la listas, como si fuera una lista enlazada, recorriendo su anterior.

- Tipos de Datos: para el manejo de los tipos de datos se realizo una clase enumerada, que contendria los tipos de datos int, double, char, string, etc. Esta tambien contendria todas las operaciones relacionales, aritmeticas y logicas.

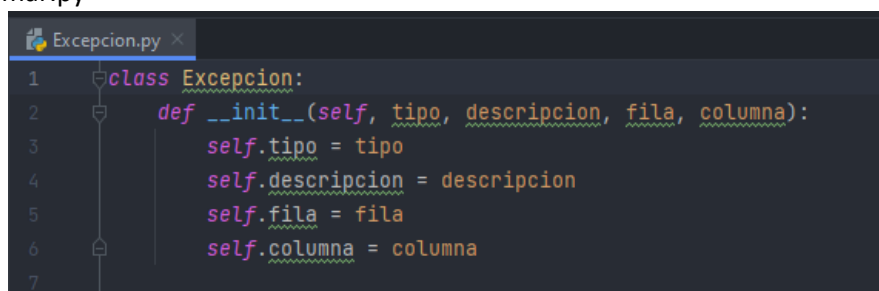


```

Tipo.py
1 from enum import Enum
2
3
4 class TIPO(Enum):
5     ENTERO = 1
6     DECIMAL = 2
7     BOOLEANO = 3
8     CARACTER = 4
9     CADENA = 5
10    NULO = 6
11    ARREGLO = 7

```

- Manejo de errores (excepciones): Para el manejo de errores se realizo una clase "Excepcion", estos objetos seran almacenados dentro de la lista de errores general que utilizamos en nuestra clase grammar.py



```

Excepcion.py
1 class Excepcion:
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

```

Realizacion de Arbol AST.

Para la realizacion del arbol de necesita de una clase donde se puedan ir almacenando los nodos y la informacion que tenemos como errore y funciones.

```
class Arbol:
    def __init__(self, instrucciones):
        self.instrucciones = instrucciones
        self.__funciones = []
        self.excepciones = []
        self.consola = ""
        self.TSglobal = None
        self.__texto_interfaz = None
        self.dot = ""
        self.contador_dot = 0
```

En esta parte se utilizo la clase NodoArbol que es una clase que nos sirve para almacenar los hijos de los nodo padre que vamos a declarar. En este caso para cada instrucción se le implementa el metodo de getNodo() para asi poder ir creando los nodos.

- Instrucción Print: Por ejemplo para la clase “Imprimir” de las Instrucciones se debe de implementar el getNodo de la forma en la que tengamos un Nodo PRINT y un Nodo para la expresion.

```
def getNodo(self):
    nodo = NodoArbol("PRINT")
    nodo.addHijoNodo(self.expresion.getNodo())
    return nodo
```

- Instrucción For: Para la instrucción for se debe de implementar el getNodo() pero de una manera en la que recorramos nuestras instrucciones, ya que la gramatica construida tiene una lista de instrucciones. Realizando un nodo para cada instrucción y llamando al getNodo() de las instrucciones.

```
def getNodo(self):
    nodo = NodoArbol("FOR")
    nodo.addHijoValor("(")
    nodo.addHijoNodo(self.declara_asigna.getNodo())
    nodo.addHijoNodo(self.expresion.getNodo())
    nodo.addHijoNodo(self.uno_en_uno.getNodo())
    nodo.addHijoValor(")")

    nodo_instrucciones = NodoArbol("INSTRUCCIONES FOR")

    for instruccion in self.instrucciones:
        nodo_instrucciones.addHijoNodo(instruccion.getNodo())
    nodo.addHijoNodo(nodo_instrucciones)
    return nodo
```

****** Para comprender esto, hay que comprender que las expresiones tienen su propio metodo getNodo() por eso lo definimos en cada una de las clases Intruccion y de las clases Expresion, para que nos retorne un nodo ya realizado en caso de tener un a lista de isntrucciones.******

- Metodo Recorrer: El metodo `recorreAST` recorre el ast desde el nodo padre, para poder obtener el valor de los nodos y asi realizar la grafica en graphviz.

```
def recorrerAST(self, idPadre, nodoPadre):
    for hijo in nodoPadre.getListaHijos():
        nombreHijo = "n" + str(self.contador_dot)
        self.dot += nombreHijo + "[label=\"" + hijo.getValor().replace("\", "\\\"") + "\"];\n"
        self.dot += idPadre + "->" + nombreHijo + ";\n"
        self.contador_dot += 1
        self.recorrerAST(nombreHijo, hijo)
```

- Metodo `getDot`: Obtiene la cadena para realizar el ast en graphviz.

```
def getDot(self, raiz): # Retorna la cadena de la grafica
    self.dot = ""
    self.dot += "digraph {\n" \
        "bgcolor=\"#F2F4F4\"; node[style=bold, color=\"#27AE60\", style=\"filled,setlinewidth(2)\", \" \" \
        \"fillcolor=white];\n"
    self.dot += "n0[label=\"" + raiz.getValor().replace("\", "\\\"") + "\"];\n"
    self.contador_dot = 1
    self.recorrerAST("n0", raiz)
    self.dot += "}\n"
    return self.dot
```

- Para realizar ya el pdf en graphviz se hace un nodo padre llamado Raiz y que tenga un nodo llamado Instrucciones, posteriormente se recorre el arbol con el metodo que nos devuelve la cadena y con un comando del sistema generamos el pdf.

```
def realizar_dot(astTree):
    inicio_dot = NodoArbol("RAIZ")
    instruccion_dot = NodoArbol("INSTRUCCIONES")

    for instruccion_ast in astTree.getInstrucciones():
        instruccion_dot.addHijoNodo(instruccion_ast.getNodo())

    inicio_dot.addHijoNodo(instruccion_dot)
    grafo = astTree.getDot(inicio_dot) # Devuelve el string

    dirname = os.path.dirname(__file__)
    direcc = os.path.join(dirname, 'ast.dot')
    file = open(direcc, "w+")
    file.write(grafo)
    file.close()
    os.system('dot -T pdf -o ast.pdf ast.dot')
```

Realizar Analisis:

Para realizar el analisis se hace un metodo en el que se vaya haciendo recorridos a las instrucciones, agregando las funciones, declaraciones y asignaciones requeridas. Tomando en cuenta que tenemos que guardar en nuestro arbol las declaraciones y funciones que esten fuera del main.

```
def grammar_analisis(entrada, txtWidget):
    # ----- EXPORTACIONES -----
    from Tabla_Simbolo.Arbol import Arbol
    from Tabla_Simbolo.TablaSimbolos import TablaSimbolos

    """
    El parse lo que nos devuelve es un arbol con nodos
    Hasta ahora solo hemos hecho el analisis sintactico
    """

    instrucciones = parse(entrada)  # ARBOL AST
    astTree = Arbol(instrucciones)  # Inicializamos nues
    Tabla_Global = TablaSimbolos()  # Inicializamos la t
    astTree.setSglobal(Tabla_Global)
    astTree.setTextoInterfaz(txtWidget)  # Agregamos el widge
    creacion_nativas(astTree)  # Creando funciones

    # CAPTURA DE ERRORES LEXICOS Y SINTACTICOS
    for error in errors:
        astTree.getExcepciones().append(error)
        astTree.updateConsola(error.toString())
```

Lenguaje JPR:

Para la realizacion del lenguaje definido JPR se uso la librería PLY de python 3.8. Realizando el análisis léxico y sintáctico de esta manera. A continuacion de mostraré la manera en la que se trabajaron los distintos analisis dentro de este lenguaje.

Para el manejo de errores se hizo una lista general de errores y se manejo con objetos de tipo "Excepcion" que contienen el tipo de error, la descripcion, la linea y columna del error.

```
1 class Excepcion:
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
```

Análisis Léxico:

Primeramente para tener un buen uso de palabras reservadas, estas se almacenaron dentro de un diccionario de python, teniendo una {llave, valor} que nos permitiria acceder a las palabras reservadas mas facilmente.

```

13 reserved = {
14     'var': 'RVAR',
15     'new': 'RNEW',
16     'print': 'RPRINT',
17
18     # Tipos de datos primitivos
19     'false': 'RFALSE',
20     'true': 'RTRUE',
21     'null': 'RNULL',
22

```

Tambien se uso una lista de tokens que contienen todos los tokens definidos por el lenguaje JPR. Proximamente se le adjunta el diccionario que creamos con las palabras reservadas.

```

tokens = [
    # Comentarios
    'COMMENTUNLINE', # Comentario Unilinea
    'COMMENTLINES', # Comentario multilinea
    'ID', # Ident
    ] + list(reserved.values())

```

Para los tokens que no son palabras reservadas, se les necesita agregar un patron, por lo que se definen según **PLY** cada uno de sus patrones, esto definiendo el simbolo y haciendo uso de expresiones regulares.

```

# Arreglos
t_CLASPSYMBOLOPEN = r'\['
t_CLASPSYMBOLCLOSE = r'\]'

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

```

Las expresiones regulares usadas son las siguientes:

- **t_DECIMAL = '\d+\.\d+'** expresion regular que evalua si el token es decimal, pudiendo venir cualquier digito una o mas veces, seguido de punto, seguido de digito una o mas veces.
- **t_ID = '[a-zA-Z_][a-zA-Z_0-9]*'** expresion que evalua si el token reconocido es un ID. Algo hay que aclarar de esto es que los tokens ID y palabras reservadas se evaluan con esta expresion misma, ya que se evalua si el ID reconocido es una palbara reservada de nuestro diccionario definido con anterioridad.
- **t_CADENA = r'""\"(\\\"|\\\\|\\|\\n|\\t|\\r|[^\\\"\\\\])*?\\\"'** expresion que evalua si el token es una cadena. En el lenguaje eta permitido colocar algo como lo siguiente: "\\\" por lo que esta expresion evalua si solamente viene el simbolo "\\\" solo, entonces la cadena reconocida seria un error.

- **t_CHARACTER** = `r"\"'` Expresion que evalua si el token es un carácter, igualmente que con la cadena se evalua si el símbolo “\” viene solo. Recalcando para las cadenas y caracteres, se hizo un reemplazo en el token para reemplazar los simbolos de escape del lenguaje JPR ya que en el texto del token un “\n” se definiria como tal y no como un salto de linea.
- **t_COMMENTLINES** = `r'\#*(.|\\n)*?\\#'` Expresion que evalua si el token es un comentario multilinea.
- **t_COMMENTUNLINE** = `r'\#.*\\n'` Expresion que evalua si es un comentario unilinea.
- **Errores:** Para los errores lexicos se tuvo que definir una produccion **t_error** teniendo como retorno un objeto de clase Excepcion y agregandolo a la lista de errores.

Los demas tokens eran simbolos o patrones ya definidos, con lo cual fueron colocados directamente.

Analisis Sintactico

Para la realizacion del analisis sintactico se uso una gramatica ascendente recursiva por la izquierda, esto con el proposito de ayudar a nuestra herramienta a no tener conflictos, es posible por que la herramienta **PLY** construye el analizador en base al metodo LALR de analisis sintactico.

Los aspectos importantes a considerar y que se trabajaron son lo ssiguientes:

- **Precedencia:** Se uso precedencia en los operadores para poder evaluar las expresiones de manera correcta, según lo definido por el lenguaje JPR.

```
# Asociación de operadores y precedencia
precedence = (
    ('left', 'OR'), # ||
    ('left', 'AND'), # &&
    ('right', 'UNOT'), # !
    ('left', 'EQUALIZATIONSIGN', 'DIFFERENTIATIONSIGN', 'SMALLERTHAN', 'LESSEQUAL', 'GREATERTHAN', 'GREATEREQUAL'),
    ('left', 'PLUSSIGN', 'SUBTRACTIONSIGN'), # + -
    ('left', 'MULTIPLICATIONSIGN', 'DIVISIONSIGN', 'MODULESIGN'), # / * %
    ('left', 'POWERSIGN'), # **
    ('right', 'UMENOS'), # -
    ('left', 'INCREMENT', 'DECREMENT'), # ++ --
)
```

Para la gramatica se utilizo una lista de instrucciones que pueden contener varias expresiones, esta gramtica sera explicada en el documento de la gramatica llamado Manual_Gramatica.pdf.


```
# ----- PRODUCCION INICIAL -----
def p_init(t):
    'init : instrucciones'
    t[0] = t[1]

def p_instrucciones_instrucciones_instruccion(t):
    'instrucciones : instrucciones instruccion'
    if t[2] != "":
        t[1].append(t[2])
    t[0] = t[1]
```

Requerimientos Computadora:

Los detalles de la computadora donde se corrió dicho programa son las siguientes:

Item	Value
OS Name	Microsoft Windows 10 Home
Version	10.0.19042 Build 19042
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-3TP8RQ3
System Manufacturer	Dell Inc.
System Model	Inspiron 5570
System Type	x64-based PC
System SKU	0810
Processor	Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz, 1992 Mhz, 4 Core(s), 8 Logical Pr...
BIOS Version/Date	Dell Inc. 1.2.3, 5/15/2019
SMBIOS Version	3.0
Embedded Controller Version	255.255
BIOS Mode	UEFI
BaseBoard Manufacturer	Dell Inc.
BaseBoard Product	0D65FD
BaseBoard Version	A00
Platform Role	Mobile
Secure Boot State	On
PCR7 Configuration	Elevation Required to View
Windows Directory	C:\WINDOWS
System Directory	C:\WINDOWS\system32
Boot Device	\Device\HarddiskVolume1
Locale	United States

Hardware Abstraction Layer	Version = "10.0.19041.964"
User Name	DESKTOP-3TP8RQ3\Isaac
Time Zone	Central America Standard Time
Installed Physical Memory (RAM)	12.0 GB
Total Physical Memory	11.9 GB
Available Physical Memory	3.50 GB
Total Virtual Memory	23.4 GB
Available Virtual Memory	12.1 GB
Page File Space	11.5 GB
Page File	C:\pagefile.sys
Kernel DMA Protection	Off
Virtualization-based security	Not enabled
Device Encryption Support	Elevation Required to View
Hyper-V - VM Monitor Mode E...	Yes
Hyper-V - Second Level Addres...	Yes
Hyper-V - Virtualization Enable...	Yes
Hyper-V - Data Execution Prote...	Yes