Andhy Lizandro Solís Osorio

201700886 |

Manual Técnico

Proyecto 2 Compiladores 2

**Minor-C**

**Descripción:**

Es la implementación de un lenguaje con las características básicas de C que se traduce a un código de más bajo nivel, en este caso AUGU el proyecto fue desarrollado en Python usando como interfaz gráfica Pyqt, el proyecto fue publicado con una licencia GNU GPL 3.

**Links Importantes:**

**Enlace de Augus** <https://es.wikipedia.org/wiki/3APHP>

**Enlace del repositorio MinorC** <https://github.com/esnorki2008/MinorC>

**Enlace de la licencia de la implementación** <https://www.gnu.org/licenses/gpl-3.0.html>

La presente implementación fue realizada con:

* Windows como sistema Operativo
* Versión de Python utilizada 3.8
* Pyqt como interfaz grafica
* Visual Studio Code con el plugin de Python como IDE
* Ply como analizador de las cadenas de entrada

**Ply**

Es una implementación en Python del famoso lex y Yacc, es capaz de generar analizadores LALR (1) lo cual lo hace un analizador capaz de manejar una amplitud de gramáticas

**Enlace de Ply** <https://www.dabeaz.com/ply/>.

**Analizador léxico**

Es el que se encarga de analizar la entrada y generar una serie de tokens en base a expresiones regulares y palabras reservadas, estos tokens son reconocidos por el analizador sintáctico quien los espera en un cierto orden para determinar si la cadena es la correcta.

En caso de que venga un símbolo que no fue definido entre los permitidos se reportara el error en las interfaces graficas del proyecto.

Los tokens se definen a través de expresiones regulares y palabras reservadas, las cuales ply se encarga de analizar.

**Definición de los tokens para las palabras reservadas**

reserved = {

    'scanf': 'SCANF',

    'printf': 'PRINTF',

    'int': 'INT',

    'float': 'FLOAT',

    'char': 'CHAR',

    'double': 'DOUBLE',

    'while': 'WHILE',

    'do': 'DO',

    'struct': 'STRUCT',

    'if': 'IF',

    'else': 'ELSE',

    'switch': 'SWITCH',

    'return': 'RETURN',

    'default': 'DEFAULT',

    'case': 'CASE',

    'for': 'FOR',

    'break': 'BREAK',

    'continue': 'CONTINUE',

    'goto': 'GOTO',

    'void': 'VOID',

    'sizeof': 'SIZEOF'

}

Las palabras reservadas se crean en un diccionario y luego en la producción se determina cuales son reservadas y cuales son identificadores

def t\_IDENTIFICADOR(t):

    r'[a-zA-Z\_][a-zA-Z\_0-9]\*'

    t.type = reserved.get(t.value, 'IDENTIFICADOR')  # Check for reserved words

    return t

la producción se encarga de verificar las palabras reservadas y genera el token palabra reservada pero si detecta que la palabra reservada se encuentra entre el diccionario de las palabras reservadas retorna el respectivo token.

**Definición de los tokens individuales**

Los tokens individuales se definen con una expresión regular simple para cada token, los tokens se usaran en el analizador sintáctico

t\_IGUAL = r'\='

t\_PUNTO = r'\.'

t\_MASIGUAL = r'\+\='

t\_MENOSIGUAL = r'\-\='

t\_PORIGUAL = r'\\*\='

t\_DIVIGUAL = r'\/\='

t\_MODIGUAL = r'\%\='

t\_SHIFTIIGUAL = r'\<\<\='

t\_SHIFTDIGUAL = r'\>\>\='

t\_ANDIGUAL = r'\&\='

t\_XORIGUAL = r'\^\='

t\_ORIGUAL = r'\|\='

t\_PARA = r'\('

t\_PARC = r'\)'

**Definición de los tokens con más complejidad**

De igual manera a la definición simple se basa en una expresión regular, solo que en este caso al estar dentro de una función se puede llegar a retornar el valor “real” es decir que si viene un numero en la cadena se procede a convertirlo en un número, en caso de que de una excepción se le asigna un valor genérico de 0 al token.

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

**Definición de los errores léxicos**

Si se encuentra un error léxico Ply hace que se ejecute la función t\_error la que se encarga de manejar el símbolo no definido, se le indica al analizador “t.lexer.skip” que quiere decir que se avanzara el analizador sintáctico un token para así continuar con el análisis.

def t\_error(t):

    from AProyecto2.Contenido.Analizadores.MinorCSintactico import entrada,list\_er

    #print("error lexico ")

    #print([ord(c) for c in str(t.value[0])])

    tp=(find\_column(entrada,t))

    from AProyecto2.Contenido.TablaDeSimbolos import Error

    list\_er.append(Error("Error Analizando","No se esperaba el token "+str(t.value[0]),2,tp))

    t.lexer.skip(1)

**Analizador Sintáctico**

Es el que se encarga de construir una estructura de datos como salida en base a su entrada, en base a sus reglas semánticas se puede definir que hacer al final de cada producción en base a la serie de datos que llevaron a esa entrada, Ply tiene la opción de colocar la procedencia para los terminales que se deseen.

**Precedencia**

precedence = (

    ('right', 'INTERROGACION', 'DOBLEPUNTO'),

    ('left', 'OR'),

    ('left', 'AND'),

    ('left', 'ORB'),

    ('left', 'XORB'),

    ('left', 'ANDB'),

    ('left', 'IGUALDOBLE', 'DIFERENTE'),

    ('left', 'MAYOR', 'MAYORIGUAL', 'MENOR', 'MENORIGUAL'),

    ('left', 'SHIFTI', 'SHIFTD'),

    ('left', 'MAS', 'MENOS'),

    ('left', 'POR', 'DIVIDIDO', 'MOD'),

    ('right', 'UMENOS', 'UMAS', 'NOT', 'NOTB','UAND','MASDOBLE','MENOSDOBLE','CASTINT','CASTDOUBLE','CASTFLOAT','CASTCHAR'),

)

Los terminales de menor precedencia van en la parte superior de la tupla, mientras que los de mayor precedencia van hasta el final de la definición de la tupla, existen dos tipos de asociatividades Izquierda y Derecha, la asociatividad por la derecha regularmente se utiliza en operadores unarios o similares, mientras que en la asociatividad por la izquierda se utilizan en operadores binarios.

**Explicación De La Gramática**

def p\_inicio(t):

    'inicio    : lista\_metodos'

    t[0] = t[1]

    global min\_lst\_repga

    repg\_inst='inicio : lista\_metodo'

    repga\_sema="t[0]=t[1]"

    min\_lst\_repga.append((repg\_inst,repga\_sema))

Es la produccion inicial donde la salida definida por t[0] será utilizada para ejecutar la traducción, su produccion es lista\_metodos que es una serie de métodos e instrucciones.

def p\_metodos\_lista(t):

    'lista\_metodos : lista\_metodos metodos'

    t[0]=t[1]

    t[0].agregar(t[2])

    global min\_lst\_repga

    repg\_inst = 'lista\_metodos : lista\_metodos metodos'

    repga\_sema = '''t[0]=t[1]

    t[0].agregar(t[2])'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta produccion se encarga de retornar una serie de métodos donde, para que el analizador haya llegado a esta produccion ha de existir un método que fue reducido, por lo tanto hay una lista y se agrega el nuevo método detectado a la lista de métodos que se traía

def p\_metodos\_lista\_solo(t):

    'lista\_metodos :  metodos'

    t[0]=ListaMetodos([t[1]])

    global min\_lst\_repga

    repg\_inst = 'lista\_metodos :  metodos'

    repga\_sema = '''t[0]=ListaMetodos([t[1]])'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

Es la primera produccion de los métodos aquí se crea una lista y se procede a agregar un método a la lista

def p\_declaracion\_var\_glonal(t):

    'metodos : instruccion'

    t[0] = t[1]

    global min\_lst\_repga

    repg\_inst = 'metodos : instruccion'

    repga\_sema = '''t[0] = t[1]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

Esta produccion permite que existan instrucciones de forma global, la instrucción se retorna para ser agregada en la lista de métodos

def p\_declaracion\_struct\_a(t):

    'metodos : STRUCT IDENTIFICADOR LLAA muhco\_decla\_struct LLAC PUNTOCOMA'

    tp = find\_column(entrada, t.slice[1])

    t[0] = DefinirStruct(t[2], t[4], tp)

    global min\_lst\_repga

    repg\_inst = 'metodos : STRUCT IDENTIFICADOR LLAA muhco\_decla\_struct LLAC'

    repga\_sema = '''tp = find\_column(entrada, t.slice[1])

    t[0] = DefinirStruct(t[2], t[4], tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta instruccion permite la definición de los structs para asi saber que valores componen a los structs

def p\_instruccion\_declaracion\_struct\_lista(t):

    'muhco\_decla\_struct : muhco\_decla\_struct decla\_struct'

    t[0]=t[1]

    t[0].append(t[2])

    global min\_lst\_repga

    repg\_inst = 'muhco\_decla\_struct : muhco\_decla\_struct decla\_struct'

    repga\_sema = '''t[0]=t[1]

    t[0].append(t[2])'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

es una lista de los valores que componen los structs, se guarda su tipo para asi saber que tipos son permitidos asignarle a ese atributo del struct.

def p\_instruccion\_declaracion\_struct\_epsilon(t):

    'muhco\_decla\_struct : decla\_struct'

    t[0]=[t[1]]

    global min\_lst\_repga

    repg\_inst = 'muhco\_decla\_struct : decla\_struct'

    repga\_sema = '''t[0]=[t[1]]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

es el primer terminal de la lista de contenido para el struct, se crea una lista y se le agrega el primer contenido del struct

def p\_instruccion\_declaracion\_struct\_puntual(t):

    'decla\_struct : tipo mdecla  corche PUNTOCOMA'

    tp = find\_column(entrada, t.slice[4])

    t[0]=ContenidoStruct(t[1],t[2],tp,t[3])

    global min\_lst\_repga

    repg\_inst = 'decla\_struct : tipo mdecla  PUNTOCOMA'

    repga\_sema = '''tp = find\_column(entrada, t.slice[3])

    t[0]=ContenidoStruct(t[1],t[2],tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

este metodo permite declarer en un struct variables simples

def p\_metodos\_declara(t):

    'metodos : tipo IDENTIFICADOR PARA parametros PARC instruccion'

    tp = find\_column(entrada, t.slice[2])

    t[0]=FuncMetodo(t[2],t[4],t[6],tp)

    global min\_lst\_repga

    repg\_inst = 'metodos : tipo IDENTIFICADOR PARA parametros PARC instruccion'

    repga\_sema = '''tp = find\_column(entrada, t.slice[2])

    t[0]=FuncMetodo(t[2],t[4],t[6],tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

este metodo permite delcarar loso métodos con sus respectivos parámetros y una lista de instrucciones como cuerpo

def p\_metodos\_declara\_parametros\_muchos(t):

    'parametros : parametros COMA tipo IDENTIFICADOR'

    t[0]=t[1]

    tupla = (t[3], t[4])

    t[0].append(tupla)

    global min\_lst\_repga

    repg\_inst = 'parametros : parametros COMA tipo IDENTIFICADOR'

    repga\_sema = '''t[0]=t[1]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

estea produccion permite declarer muchos parámetros en el método

def p\_metodos\_declara\_parametros\_solo(t):

    'parametros : tipo IDENTIFICADOR'

    t[0]=[]

    tupla = (t[1], t[2])

    t[0].append(tupla)

    global min\_lst\_repga

    repg\_inst = 'parametros : tipo IDENTIFICADOR'

    repga\_sema = '''t[0]=[]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

es la primera produccion a la cual reduce el analizador si se encuentra por lo menos un parámetro

def p\_metodos\_declara\_parametros\_solo\_va(t):

    'parametros : '

    t[0]=[]

    global min\_lst\_repga

    repg\_inst = 'parametros : '

    repga\_sema = '''t[0]=[]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

si no se enecuentra ningun solo parámetro se deriva a esta produccion

def p\_instruccion(t):

    'instrucciones : instrucciones instruccion '

    t[0] = t[1]

    t[0].agregar(t[2])

    global min\_lst\_repga

    repg\_inst = 'instrucciones : instrucciones instruccion '

    repga\_sema = '''t[0] = t[1]

    t[0].agregar(t[2])'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta produccion indica una lista de instrucciones que regularmente son las que van adentro del contenido de los métodos

def p\_instrucciones\_inicio(t):

    'instrucciones : instruccion '

    t[0] = ListaInstruccion([t[1]])

    global min\_lst\_repga

    repg\_inst = 'instrucciones : instruccion '

    repga\_sema = '''t[0] = ListaInstruccion([t[1]])'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

es el inicio de la lista de instrucciones, se crea una lista y se agrega el primer elemento de la lista en la construcción de la misma

def p\_sentencia\_etiqueta\_definicion(t):

    '''instruccion : IDENTIFICADOR DOBLEPUNTO'''

    tp = find\_column(entrada, t.slice[1])

    t[0]=FuncLabel(t[1],tp)

esta produccion sirve para definir una etiqueta dentro del contenido

def p\_sentencia\_etiqueta\_goto(t):

    '''instruccion : GOTO IDENTIFICADOR PUNTOCOMA'''

    tp = find\_column(entrada, t.slice[1])

    t[0] = Goto(t[2], tp)

esta produccion sirve para colocar un goto hacia una etiqueta

def p\_sentencia\_encapsulado(t):

    '''instruccion : LLAA  instrucciones LLAC'''

    t[0]=t[2]

    global min\_lst\_repga

    repg\_inst = '''instruccion : LLAA  instrucciones LLAC'''

    repga\_sema = '''t[0]=t[2]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta produccion permite tener una lista de instrucciones dentro de llaves, regularmente son utilizados después de una estructura de control o la definición de un método

def p\_sentencia\_encapsulado\_vacia(t):

    '''instruccion : LLAA   LLAC'''

    t[0] = ListaInstruccion([])

    global min\_lst\_repga

    repg\_inst = '''instruccion : LLAA   LLAC'''

    repga\_sema = '''t[0] = ListaInstruccion([])'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

Tambien se puede definer una sentencia con solamente las llaves y no instrucciones internamente

def p\_instruccion\_ciclo(t):

    'instruccion : instruccion\_ciclo'

    t[0]=t[1]

    global min\_lst\_repga

    repg\_inst = 'instruccion : instruccion\_ciclo'

    repga\_sema = '''t[0]=t[1]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta produccion es usada por los ciclos, solamente se encarga de ascender el valor del ciclo que fue reducido con anterioridad

def p\_instruccion\_abierta\_inst(t):

    'instruccion : instruccion\_abierta'

    t[0]=t[1]

    global min\_lst\_repga

    repg\_inst = 'instruccion : instruccion\_abierta'

    repga\_sema = '''t[0]=t[1]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta instruccion es utilizada para disminuir la cantidad de shift reduce que puede ocasionar la sentencia if else, se encarga de ascender el valor que fue reducido antes

def p\_instruccion\_llamado\_metodos(t):

    'instruccion : instruccion\_llamar\_metodo'

    t[0]=t[1]

    global min\_lst\_repga

    repg\_inst = 'instruccion : instruccion\_llamar\_metodo'

    repga\_sema = '''t[0]=t[1]'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta instruccion sirve para agregar a la lista de métodos

def p\_interna\_break(t):

    'instruccion\_llamar\_metodo : BREAK  PUNTOCOMA'

    tp = find\_column(entrada, t.slice[1])

    t[0]=FuncBreak(tp)

    global min\_lst\_repga

    repg\_inst = 'instruccion\_llamar\_metodo : BREAK  PUNTOCOMA'

    repga\_sema = '''tp = find\_column(entrada, t.slice[1])

    t[0]=FuncBreak(tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta instruccion se encarga de terminar la instrucción de ciclo que la debería de contener

def p\_interna\_continue(t):

    'instruccion\_llamar\_metodo : CONTINUE  PUNTOCOMA'

    tp = find\_column(entrada, t.slice[1])

    t[0]=FuncContinue(tp)

    global min\_lst\_repga

    repg\_inst = 'instruccion\_llamar\_metodo : CONTINUE  PUNTOCOMA'

    repga\_sema = '''tp = find\_column(entrada, t.slice[1])

    t[0]=FuncContinue(tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

la estructura continue permite pasar a la siguiente iteración del ciclo

def p\_llamar\_retorno\_metodo(t):

    'instruccion\_llamar\_metodo : RETURN expresiones PUNTOCOMA'

    tp = find\_column(entrada, t.slice[1])

    t[0]=FuncReturn(t[2],tp)

    global min\_lst\_repga

    repg\_inst = 'instruccion\_llamar\_metodo : RETURN expresiones PUNTOCOMA'

    repga\_sema = '''tp = find\_column(entrada, t.slice[1])

    t[0]=FuncReturn(t[2],tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

esta produccion sirve para definer el retorno de un método

def p\_llamar\_funcion\_generica(t):

    'instruccion\_llamar\_metodo : IDENTIFICADOR PARA listado\_parametros PARC PUNTOCOMA'

    tp = find\_column(entrada, t.slice[1])

    t[0] = LlamarFuncMetodo(t[1],t[3],tp)

    global min\_lst\_repga

    repg\_inst = 'instruccion\_llamar\_metodo : IDENTIFICADOR PARA listado\_parametros PARC PUNTOCOMA'

    repga\_sema = '''tp = find\_column(entrada, t.slice[1])

    t[0] = LlamarFuncMetodo(t[1],t[3],tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

se encarga de definer el llamado hacia una función con una serie de parámetros

def p\_llamar\_printf\_algo(t):

    'instruccion\_llamar\_metodo : PRINTF PARA listado\_parametros PARC PUNTOCOMA'

    tp = find\_column(entrada, t.slice[1])

    t[0] = FuncPrintF(t[3],tp)

    global min\_lst\_repga

    repg\_inst = 'instruccion\_llamar\_metodo : PRINTF PARA listado\_parametros PARC PUNTOCOMA'

    repga\_sema = '''tp = find\_column(entrada, t.slice[1])

    t[0] = FuncPrintF(t[3],tp)'''

    min\_lst\_repga.append((repg\_inst, repga\_sema))

sirve para utilizer la function printf

def p\_expresiones\_unarias(t):

    '''expresiones : MENOS expresiones %prec UMENOS

                    | MAS expresiones %prec UMAS

                    | NOTB expresiones

                    | ANDB expresiones %prec UAND

                    | NOT expresiones '''

Se encarga de definir las expresiones que tienen un solo operador

def p\_expresiones\_binarias(t):

    '''expresiones : expresiones OR expresiones

                    | expresiones AND expresiones

                    | expresiones ORB expresiones

                    | expresiones XORB expresiones

                    | expresiones ANDB expresiones

                    | expresiones IGUALDOBLE expresiones

                    | expresiones DIFERENTE expresiones

                    | expresiones MAYOR expresiones

                    | expresiones MAYORIGUAL expresiones

                    | expresiones MENOR expresiones

                    | expresiones MENORIGUAL expresiones

                    | expresiones SHIFTI expresiones

                    | expresiones SHIFTD expresiones

                    | expresiones MAS expresiones

                    | expresiones MENOS expresiones

                    | expresiones POR expresiones

                    | expresiones MOD expresiones

                    | expresiones DIVIDIDO expresiones'''

    global entrada

    tp = find\_column(entrada, t.slice[2])

    t[0] = NodoBinario(t[1], t[2], t[3], tp)

se encarga de definir las operaciones que están compuestas de dos nodos y un operador en el medio

**Producción de error**

def p\_error(t):

    from AProyecto2.Contenido.Analizadores.MinorCSintactico import list\_er,entrada

    #print(id(list\_er))

    if t is None:

        from AProyecto2.Contenido.TablaDeSimbolos import Error

        tp =(0,0)

        list\_er.append(Error("Error Analizando","Se llego al fin del archivo",1,tp))

    else:

        tp = find\_column(entrada, t)

        #print("Error sintáctico en '%s'" % t)

        from AProyecto2.Contenido.TablaDeSimbolos import Error

        list\_er.append(Error("Error Analizando","No se esperaba el token "+str(t.value),1,tp))

si se encuentra un error durante el análisis se procede a avanzar un token y se guarda en una lista de errores el token que hacia falta

**Como Fue Implementado**

Se implemento utilizando el patrón interprete, el cual consiste en una clase abstracta como padre de otras clases, es un patrón sencillo de utilizar para el análisis de lenguajes

Enlace de referencia: <https://programacion.net/articulo/patrones_de_diseno_xvi_patrones_de_comportamiento_interpreter_1019>

**Interprete de Augus Utilizado Junto Al Traductor De Minor C**

<https://github.com/harias25/Augus>