Manual de Tecnico

Flujo de la aplicación

El flujo se describe de la siguiente manera:

1. Se ingresa el código en MinorC ya sea por archivo o editado.
2. Se ejecuta el código con el menú y/o con un ícono (o paso a paso).
3. Se transforma el código de MinorC en código de Augus, como representación intermedia. (tomar en cuenta utilizar las técnicas descritas en el libro de texto,
como la de backpatch, los GDA los cuales podrían generar código de tres direcciones distinto)
4. Se optimiza el código de Augus. (la optmización basada en las reglas de bloques y mirilla descritas en el libro de texto)
5. El resultado de la ejecución se muestra en la consola.
6. Opcionalmente se pueden acceder a los reportes.

Librerias Pyqt5,PLY, Graphviz

MinorC es un subconjunto del lenguaje C, creado con el fin de poner en práctica los conceptos del proceso de compilación, para cubrir las competencias del curso

Gramatica

```
reservadas = {
    'printf' : 'IMPRIMIR',
    'mientras' : 'MIENTRAS',
    'abs' : 'ABS',
    'unset' : 'UNSET',
    'int' : 'INT',
    'float' : 'FLOAT',
    'char' : 'CHAR',
    'void' : 'VOID',
    'if' : 'IF',
    'xor' : 'XOR',
    'array' : 'ARRAY',
    'goto' : 'GOTO',
    'exit' : 'EXIT',
    'auto' : 'AUTO',
    'break' : 'BREAK',
    'case' : 'CASE',
    'continue' : 'CONTINUE',
    'default' : 'DEFAULT',
    'do' : 'DO',
    'double' : 'DOUBLE',
```

```
    'else' : 'ELSE',
    'enum' : 'ENUM',
    'extern' : 'EXTERN',
    'for' : 'FOR',
    'register' : 'REGISTER',
    'return' : 'RETURN',
    'sizeof' : 'SIZEOF',
    'switch' : 'SWITCH',
    'struct' : 'STRUCT',
    'void' : 'VOID',
    'while' : 'WHILE',
    'return' : 'RETURN'

}

tokens  = [
    'PTCOMA',
    'COMA',
    'DOSPUNTOS',
    'LLAVIZQ',
    'LLAVDER',
    'PARIZQ',
    'PARDER',
    'CORIZQ',
    'CORDER',
    'IGUAL',
    'MAS',
    'MENOS',
    'MENOSMENOS',
    'POR',
    'DIVIDIDO',
    'CONCAT',
    'MENQUE',
    'MAYQUE',
    'IGUALQUE',
    'NIGUALQUE',
    'DECIMAL',
    'ENTERO',
    'CADENA',
    'ID',
    'TEMP',
    'PARAM',
    'VAL',
    'PILA',
    'PUNTERO',
```

```
    'DIR',
    'CARACTER',
    'RESIDUO',
    'NOT',
    'NOTBIT',
    'AND',
    'ANDBIT',
    'OR',
    'ORBIT',
    'PDECIMAL',
    'PSTRING',
    'PDOUBLE',
    'PCARACTER',

    'XORBIT',
    'MENORBIT',
    'MAYORBIT',
    'MENIGUAL',
    'MAYIGUAL'

] + list(reservadas.values())

# Tokens
t_PTCOMA    = r';'
t_COMA    = r','
t_DOSPUNTOS = r':'
t_LLAVIZQ   = r'{'
t_LLAVDER   = r'}'
t_PARIZQ    = r'\('
t_PARDER    = r'\)'
t_CORIZQ    = r'\['
t_CORDER    = r'\]'
t_IGUAL     = r'='
t_MAS       = r'\+'
t_MENOS     = r'-'
t_MENOSMENOS     = r'\-\-'
t_POR       = r'\*'
t_DIVIDIDO  = r'/'
t_NOT = r'!'
t_NOTBIT = r'\~'
t_CONCAT    = r'&y'
t_AND = r'&&'
t_ANDBIT = r'&'
t_PUNTERO = r'°'
```

```
t_XORBIT = r'\^'
t_MENORBIT = r'<<'
t_MAYORBIT = r'>>'
t_OR = r'\|\|'
t_ORBIT = r'\|'
t_MENIGUAL = r'<='
t_MAYIGUAL = r'>='
t_MENQUE    = r'<'
t_MAYQUE    = r'>'
t_IGUALQUE  = r'=='
t_NIGUALQUE = r'!='

t_RESIDUO = r'%'
t_PDECIMAL =    r'%f'
t_PCARACTER =    r'%c'
t_PDOUBLE =    r'%d'
t_PSTRING =    r'%s'
import ts as TS

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_ID(t):
     r'[a-zA-Z_][a-zA-Z_0-9]*'
     t.type = reservadas.get(t.value.lower(),'ID')    # Check for reserved w
ords
     return t


def t_CADENA(t):
```

```
    r'\".*?\"'
    t.value = t.value[1:-1] # remuevo las comillas
    return t

def t_CARACTER(t):
    r'\'.*?\''
    t.value = t.value[1:-1] # remuevo las comillas
    return t

# Comentario de múltiples líneas /* .. */
def t_COMENTARIO_MULTILINEA(t):
    r'/\*(.|\n)*?\*/'
    t.lexer.lineno += t.value.count('\n')

# Comentario simple // ...
def t_COMENTARIO_SIMPLE(t):
    r'//.*\n'
    t.lexer.lineno += 1
def t_BACKLASH(t):
    r'\\.*\n'
    t.lexer.lineno += 1

# Caracteres ignorados
t_ignore = " \t"

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

def t_error(t):
    #print("Illegal character '%s'" % t.value[0])
    #print("columna",str(find_column(t.lexer.lexdata,t)))
    error = "Error lexico en el lexema: \'"+ t.value[0]+"\' la linea: " + st
r(t.lexer.lineno) + " columna: " + str(find_column(t.lexer.lexdata,t))
    lista_errores.append(error)
    t.lexer.skip(1)

# Construyendo el analizador léxico
import ply.lex as lex
lexer = lex.lex()
lista_errores = []

# Asociación de operadores y precedencia
precedence = (
    ('left', 'OR','XOR'),
```

```
    ('left', 'AND'),
    ('left', 'IGUALQUE', 'NIGUALQUE'),
    ('left', 'MENQUE', 'MAYQUE'),
    ('left', 'MAYIGUAL', 'MENIGUAL'),
    ('right','NOTBIT'),
    ('left', 'XORBIT'),
    ('left', 'ANDBIT','ORBIT'),
    ('left', 'MENORBIT','MAYORBIT'),
    ('left','MAS','MENOS'),
    ('left','POR','DIVIDIDO'),
    ('left','RESIDUO'),
    ('right','UMENOS','NOT','NOTBIT'),
    ('left', 'PARIZQ','PARDER'),
    )

# Definición de la gramática

from expresiones import *
from instrucciones import *
from anytree import Node, RenderTree
from anytree.exporter import DotExporter
from anytree.exporter import UniqueDotExporter
gramatical = []
n_init = Node("raiz")
def p_init(t) :
    'init            : instrucciones'
    t[0] = t[1]
    global gramatical
    gramatical.append( " inicio.val : metodos.val")
def p_metodos(t) :
    'metodos    : metodos definicion_metodo'
    t[1].append(t[2])
    t[0] = t[1]
    global gramatical
    gramatical.append( " metodos.val :metodos1.append(metodo.val)")

def p_metodos_metodo(t) :
    'metodos    : definicion_metodo '
    t[0] = [t[1]]
    global gramatical
    gramatical.append( " metodos.val: metodos.val")


def p_instrucciones_lista(t) :
    'instrucciones    : instrucciones instruccion'
```

```
    t[1].append(t[2])
    t[0] = t[1]
    global gramatical
    gramatical.append( " instrucciones.val : instrucciones1.append(instrucci
on.val)")

def p_instrucciones_instruccion(t) :
    'instrucciones    : instruccion '
    t[0] = [t[1]]
    global gramatical
    gramatical.append( " instrucciones.val: instruccion.val")
def p_instruccion(t) :
    '''instruccion      : imprimir_instr
                        | definicion_variable

                        | asignacion_variable
                        | definicion_metodo
                        | definicion_struct
                        | return
                        | if_instr
                        | if_else
                        | switch
                        | break
                        | while
                        | aumento
                        | decremento
                        | dowhile
                        | for
                        | etiqueta
                        | goto
                        | llamada_funcion
                        '''
    t[0] = t[1]
    global gramatical
    gramatical.append( " instruccion.val: expresiones.val")
def p_etiqueta(t):
    'etiqueta : ID DOSPUNTOS'
    t[0] = Etiqueta(t[1])
def p_goto(t):
    'goto : GOTO ID PTCOMA'
    t[0] = Goto(t[2])
def p_return(t):
    'return : RETURN expresion_numerica'
    t[0] = Return(t[2])
    global gramatical
```

```
    gramatical.append( " return.val : expresion.val")
def p_definir_struct(t):
    'definicion_struct : STRUCT ID LLAVIZQ instrucciones LLAVDER PTCOMA'
    t[0] = Definicion_Struct(t[2],t[1],t[4])
    global gramatical
    gramatical.append( " struct.val : new Struct(id.val,instrucciones.val);"
)
def p_definir_metodo(t):
    'definicion_metodo : tipo ID PARIZQ PARDER LLAVIZQ instrucciones LLAVDER
'
    t[0] = Definicion_Metodo(t[2],t[1],t[6])
    global gramatical
    gramatical.append( " metodo.val : id.val")

def p_definir_metodo_parametros(t):
    'definicion_metodo : tipo ID PARIZQ  parametros PARDER LLAVIZQ  instrucc
iones LLAVDER'
    t[0] = Definicion_Metodo_Parametro(t[1],t[2],t[4],t[7])
    global gramatical
    gramatical.append( " metodo.val : id.val")

def p_parametros(t):
    'parametros : parametros COMA parametro'
    t[1].append(t[3])
    t[0] = t[1]
    global gramatical
    gramatical.append( " parametros.val : parametros.append(parametro.val)")

def p_parametros_parametro(t):
    'parametros : parametro'
    t[0] = [t[1]]
    global gramatical
    gramatical.append( " parametros.val : parametro.val")

def p_parametro(t):
    'parametro : tipo ID'
    t[0] = Parametro(t[1],t[2])
    global gramatical
    gramatical.append( " parametro.val : new parametro(tipo.val,id.val);")

def p_llamada_funcion(t):
    'llamada_funcion : ID PARIZQ lista_ids PARDER PTCOMA'
    t[0] = Llamada_Funcion(t[1],t[3])
    global gramatical
    gramatical.append( " llamada.val : new Llamada(id.val,lista_ids.val);")
```

```
def p_instruccion_imprimir(t) :
    'imprimir_instr     : IMPRIMIR PARIZQ expresion_numerica PARDER PTCOMA'
    t[0] =Imprimir(t[3])
    global gramatical
    gramatical.append( " imprimir.val : expresion.val")
def p_print_compuesto(t):
    'imprimir_instr : IMPRIMIR PARIZQ expresion_numerica COMA lista_ids PARD
ER PTCOMA'
    t[0] = ImprimirCompuesto(t[3],t[5])
    global gramatical
    gramatical.append( " imprimir.val : expresion.val")
def p_asignacion_variable(t):
    'asignacion_variable   : ID IGUAL expresion_numerica PTCOMA'
    t[0] = Asignacion(t[1], t[3])
    global gramatical
    gramatical.append( " asignartemp.val : expresion.val")
def p_definicion_asignacion_variable(t):
    'definicion_variable : tipo lista_ids IGUAL expresion_numerica PTCOMA'
    t[0] = Definicion_Asignacion(t[1],t[2],t[4])

def p_definicion_asignacion_variable_arreglo(t):
    'definicion_variable : tipo ID dimensiones IGUAL LLAVIZQ lista_ids LLAVD
ER PTCOMA'
    t[0] = Definicion_Asignacion_Arreglo_Multiple(t[1],t[2],t[3],t[6])
    global gramatical
    gramatical.append( " definicion_variable.val : new ArregloMultiple(tipo.
val,id.val,dimensiones.val,lista_ids.val);")

def p_acceso_arreglo(t):
    'expresion_numerica : ID dimensiones'
    t[0] = AccesoArreglo(t[1],t[2])
    global gramatical
    gramatical.append("expresion.val : dimensiones.val")
def p_dimensiones_lista(t):
    'dimensiones : dimensiones dimension'
    t[1].append(t[2])
    t[0] = t[1]
    global gramatical
    gramatical.append( " dimensiones.val : dimensiones1.append(dimension.val
)")
def p_dimensiones(t)    :
    'dimensiones : dimension'
    t[0] = [t[1]]
    global gramatical
```

```
    gramatical.append( " dimensiones.val : dimension.val")
def p_dimension(t):
    'dimension : CORIZQ expresion_numerica CORDER'
    t[0]= t[2]
    global gramatical
    gramatical.append( " dimension.val : expresion.val")

def p_definicion_variable(t):
    'definicion_variable : tipo lista_ids PTCOMA'
    t[0] = Definicion(t[1],t[2])
    global gramatical
    gramatical.append( " definicion_variable.val : expresion.val")
def p_if_instr(t) :
    'if_instr           : IF PARIZQ expresion_numerica PARDER LLAVIZQ instru
cciones LLAVDER'
    t[0] =If(t[3], t[6])
    global gramatical
    gramatical.append( " if.val : expresion.val")
def p_if_else(t) :
    'if_else : IF PARIZQ expresion_numerica PARDER LLAVIZQ instrucciones LLA
VDER ELSE LLAVIZQ instrucciones LLAVDER'
    t[0] = IfElse(t[3],t[6],t[10])
    global gramatical
    gramatical.append( " ifelse.val : expresion.val")

def p_if_else_if(t) :
    'if_instr : IF PARIZQ expresion_numerica PARDER LLAVIZQ instrucciones LL
AVDER ELSE if_instr'
    t[0] = IfElseIf(t[3],t[6],t[9])
    global gramatical
    gramatical.append( " ifelse.val : expresion.val")

def p_switch(t):
    'switch : SWITCH PARIZQ expresion_numerica PARDER LLAVIZQ casos LLAVDER'
    t[0] = Switch(t[3], t[6])
    global gramatical
    gramatical.append( " switch.val : expresion.val; switch.casos : casos.va
l;")
def p_casos(t):
    'casos : casos caso'
    t[1].append(t[2])
    t[0] = t[1]
    global gramatical
    gramatical.append( " casos.val : casos1.append(caso.val);")
def p_casos_caso(t):
```

```
    'casos : caso'
    t[0] = [t[1]]
    global gramatical
    gramatical.append( " casos.val = caso.val;")

def p_caso(t):
    'caso : CASE expresion_numerica DOSPUNTOS instrucciones'
    t[0] = Caso(t[2],t[4])
    global gramatical
    gramatical.append( " caso.val = caso(expresion.val,instrucciones.val);")
def p_default(t):
    'caso : DEFAULT DOSPUNTOS instrucciones'
    t[0] = Default(t[3])
    global gramatical
    gramatical.append( " caso.val = caso(expresion.val,instrucciones.val);")

def p_while(t):
    'while : WHILE PARIZQ expresion_numerica PARDER LLAVIZQ instrucciones LL
AVDER'
    t[0] = While(t[3],t[6])
    global gramatical
    gramatical.append( " while.val = while(expresion.val,instrucciones.val);
")

def p_dowhile(t):
    'dowhile : DO LLAVIZQ instrucciones LLAVDER WHILE PARIZQ expresion_numer
ica PARDER PTCOMA'
    t[0] = DoWhile(t[3],t[7])
    global gramatical
    gramatical.append( " dowhile.val = dowhile(expresion.val,instrucciones.v
al);")

def p_for(t):
    'for : FOR PARIZQ asignacion_variable expresion_numerica PTCOMA aumento_
f PARDER LLAVIZQ instrucciones LLAVDER'
    t[0] = For(t[3],t[4],t[6],t[9])
    global gramatical
    gramatical.append( " for.val = new for(asignacion.val,expresion.val,inst
ruccion.val,instrucciones.val);")
def p_for_d(t):
    'for : FOR PARIZQ asignacion_variable expresion_numerica PTCOMA decremen
to_f PARDER LLAVIZQ instrucciones LLAVDER'
    t[0] = For(t[3],t[4],t[6],t[9])
    global gramatical
```

```
    gramatical.append( " for.val = new for(asignacion.val,expresion.val,inst
ruccion.val,instrucciones.val);")
def p_tipo(t):
    '''tipo : INT
                | FLOAT
                | DOUBLE
                | CHAR
                | VOID'''
    t[0] = t[1]

def p_lista_ids(t):
    'lista_ids : lista_ids COMA expresion_numerica'
    t[1].append(t[3])
    t[0] = t[1]
    global gramatical
    gramatical.append( " lista_ids.val : lista_ids.append(expresion.val)")
def p_lista(t):
    'lista_ids : expresion_numerica'
    t[0] = [t[1]]
    global gramatical
    gramatical.append( " lista_ids.val : expresion.val")
def p_lista_cad(t):
    'lista_ids : expresion_numerica CORIZQ CORDER'
    t[0] = [t[1]]
    global gramatical
    gramatical.append( " lista_ids_cad.val : expresion.val")
def p_expresion_binaria(t):
    '''expresion_numerica : expresion_numerica MAS expresion_numerica
                        | expresion_numerica MENOS expresion_numerica
                        | expresion_numerica POR expresion_numerica
                        | expresion_numerica DIVIDIDO expresion_numerica
                        | expresion_numerica RESIDUO expresion_numerica'''
    global gramatical
    if t[2] == '+'  :
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_ARITMETICA.MAS)
        gramatical.append( " expresion.val : expresion.val + expresion.val")
    elif t[2] == '-':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_ARITMETICA.MENOS)
        gramatical.append( " expresion.val : expresion.val - expresion.val")
    elif t[2] == '*':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_ARITMETICA.POR)
        gramatical.append( " expresion.val : expresion.val * expresion.val")
    elif t[2] == '/':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_ARITMETICA.DIVIDIDO)
        gramatical.append( " expresion.val : expresion.val  expresion.val")
```

```
    elif t[2] == '%':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_ARITMETICA.RESIDUO)
        gramatical.append( " expresion.val : expresion.val '\%' expresion.va
l")

def p_expresion_binaria_realacional(t):
    '''expresion_numerica : expresion_numerica IGUALQUE expresion_numerica
                          | expresion_numerica NIGUALQUE expresion_numerica
                          | expresion_numerica MAYIGUAL expresion_numerica
                          | expresion_numerica MENIGUAL expresion_numerica
                          | expresion_numerica MAYQUE expresion_numerica
                          | expresion_numerica MENQUE expresion_numerica'''
    global gramatical
    if t[2] == '==' :
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_RELACIONAL.IGUAL)
        gramatical.append( " expresion.val : expresion.val igualigual expres
ion.val")
    elif t[2] == '!=':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_RELACIONAL.DIFERENTE)
        gramatical.append( " expresion.val : expresion.val notigual expresio
n.val")
    elif t[2] == '>=':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_RELACIONAL.MAYORIGUAL)
        gramatical.append( " expresion.val : expresion.val mayorigual expres
ion.val")
    elif t[2] == '<=':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_RELACIONAL.MENORIGUAL)
        gramatical.append( " expresion.val : expresion.val menorigual expres
ion.val")
    elif t[2] == '>':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_RELACIONAL.MAYOR_QUE)
        gramatical.append( " expresion.val : expresion.val mayor expresion.v
al")
    elif t[2] == '<':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_RELACIONAL.MENOR_QUE)
        gramatical.append( " expresion.val : expresion.val menor expresion.v
al")

def p_expresion_binaria_logica(t):
    '''expresion_numerica : expresion_numerica AND expresion_numerica
                          | expresion_numerica OR expresion_numerica
                          | expresion_numerica XOR expresion_numerica'''
    global gramatical
    if t[2] == '&&' :
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.AND)
```

```
        gramatical.append( " expresion.val : expresion.val and expresion.val
")
    elif t[2] == '||':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.OR)
        gramatical.append( " expresion.val : expresion.val or expresion.val"
)
    elif t[2] == 'xor':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.XOR)
        gramatical.append( " expresion.val : expresion.val xor expresion.val
")

def p_expresion_binaria_bit(t):
    '''expresion_numerica : expresion_numerica ANDBIT expresion_numerica
                          | expresion_numerica ORBIT expresion_numerica
                          | expresion_numerica XORBIT expresion_numerica
                          | expresion_numerica MAYORBIT expresion_numerica
                          | expresion_numerica  MENORBIT expresion_numerica'''
    global gramatical
    if t[2] == '&'  :
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.ANDBIT)
        gramatical.append( " expresion.val : expresion.val andbit expresion.
val")
    elif t[2] == '|':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.ORBIT)
        gramatical.append( " expresion.val : expresion.val orbit expresion.v
al")
    elif t[2] == '^':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.XORBIT)
        gramatical.append( " expresion.val : expresion.val xorbit expresion.
val")
    elif t[2] == '<<':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.MENORBIT)
        gramatical.append( " expresion.val : expresion.val menorbit expresio
n.val")
    elif t[2] == '>>':
        t[0] = ExpresionBinaria(t[1], t[3], OPERACION_LOGICA.MAYORBIT)

        gramatical.append( " expresion.val : expresion.val mayorbit expresio
n.val")

def p_expresion_not(t):
    'expresion_numerica : NOT expresion_numerica'
    t[0] = ExpresionNot(t[2])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
```

```
def p_expresion_notbit(t):
    'expresion_numerica : NOTBIT expresion_numerica'
    t[0] = ExpresionNotBit(t[2])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
def p_conversion_int(t):
    'expresion_numerica : PARIZQ INT PARDER expresion_numerica'
    t[0] = ExpresionConversionInt(t[4])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
def p_conversion_float(t):
    'expresion_numerica : PARIZQ FLOAT PARDER expresion_numerica'
    t[0] = ExpresionConversionFloat(t[4])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
def p_conversion_char(t):
    'expresion_numerica : PARIZQ CHAR PARDER expresion_numerica'
    t[0] = ExpresionConversionChar(t[4])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")

def p_expresion_unaria(t):
    'expresion_numerica : MENOS expresion_numerica %prec UMENOS'
    t[0] = ExpresionNegativo(t[2], TS.TIPO_DATO.NUMERO)
    global gramatical
    gramatical.append( " expresion.val : expresion.val")

def p_expresion_agrupacion(t):
    'expresion_numerica : PARIZQ expresion_numerica PARDER'
    t[0] = t[2]
    global gramatical
    gramatical.append( " expresion.val : expresion.val")

def p_expresion_number(t):
    'expresion_numerica : ENTERO'
    t[0] = ExpresionEntero(t[1], TS.TIPO_DATO.NUMERO,t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : entero.val")
def p_expresion_decimal(t):
    'expresion_numerica : DECIMAL'
    t[0] = ExpresionEntero(t[1], TS.TIPO_DATO.FLOAT,t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : decimal.val")

def p_expresion_id_labl(t):
```

```
    'expresion_numerica   : ID'
    t[0] = ExpresionIdentificador(t[1],t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : id.val")

def p_expresion_cadena(t) :
    'expresion_numerica     : CADENA'
    t[0] = ExpresionEntero(t[1], TS.TIPO_DATO.CADENA,t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : cadena.val")
def p_expresion_caracter(t):
    'expresion_numerica : CARACTER'
    t[0] = ExpresionEntero(t[1], TS.TIPO_DATO.CARACTER,t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : caracter.val")
def p_expresion_puntero_i(t):
    'expresion_numerica : PUNTERO ID'
    t[0] = ExpresionEntero(t[2], TS.TIPO_DATO.PUNTEROI,t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : caracter.val")
def p_expresion_puntero_por(t):
    'expresion_numerica : POR ID'
    t[0] = ExpresionIdentificador(t[2],t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : caracter.val")
def p_expresion_puntero_por_por(t):
    'expresion_numerica : POR POR ID'
    t[0] = ExpresionIdentificador(t[3],t.lexer.lineno)
    global gramatical
    gramatical.append( " expresion.val : caracter.val")
def p_break(t):
    'break : BREAK PTCOMA'
    t[0] = Break(t[1])
    global gramatical
    gramatical.append( " expresion.val : break.val")

def p_aumento(t):
    'aumento : MAS MAS expresion_numerica PTCOMA'
    t[0] = Aumento(t[3])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
def p_aumento_f(t):
    'aumento_f : MAS MAS expresion_numerica'
    t[0] = Aumento(t[3])
    global gramatical
```

```
    gramatical.append( " expresion.val : expresion.val")
def p_aumento_an(t):
    'aumento :  expresion_numerica MAS MAS PTCOMA'
    t[0] = Aumento(t[1])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")

def p_decremento(t):
    'decremento : MENOSMENOS expresion_numerica PTCOMA'
    t[0] = Decremento(t[2])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
def p_decremento_f(t):
    'decremento_f : MENOSMENOS expresion_numerica  '
    t[0] = Decremento(t[2])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")


def p_decremento_an(t):
    'decremento :  expresion_numerica MENOSMENOS PTCOMA'
    t[0] = Decremento(t[1])
    global gramatical
    gramatical.append( " expresion.val : expresion.val")
def getErrores():
    #print("gramatica errores:",lista_errores)
    return lista_errores
def cleanErrores():

    del lista_errores[:]

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

def p_error(t):
    print(t)
    print("Error sintáctico en '%s'" % t.value)

    while True:
        to=parser.token()
        #print("esto trae el token siguiente: ",to.type)
        if not to or to.type == 'PTCOMA' : break

    parser.errok()
```

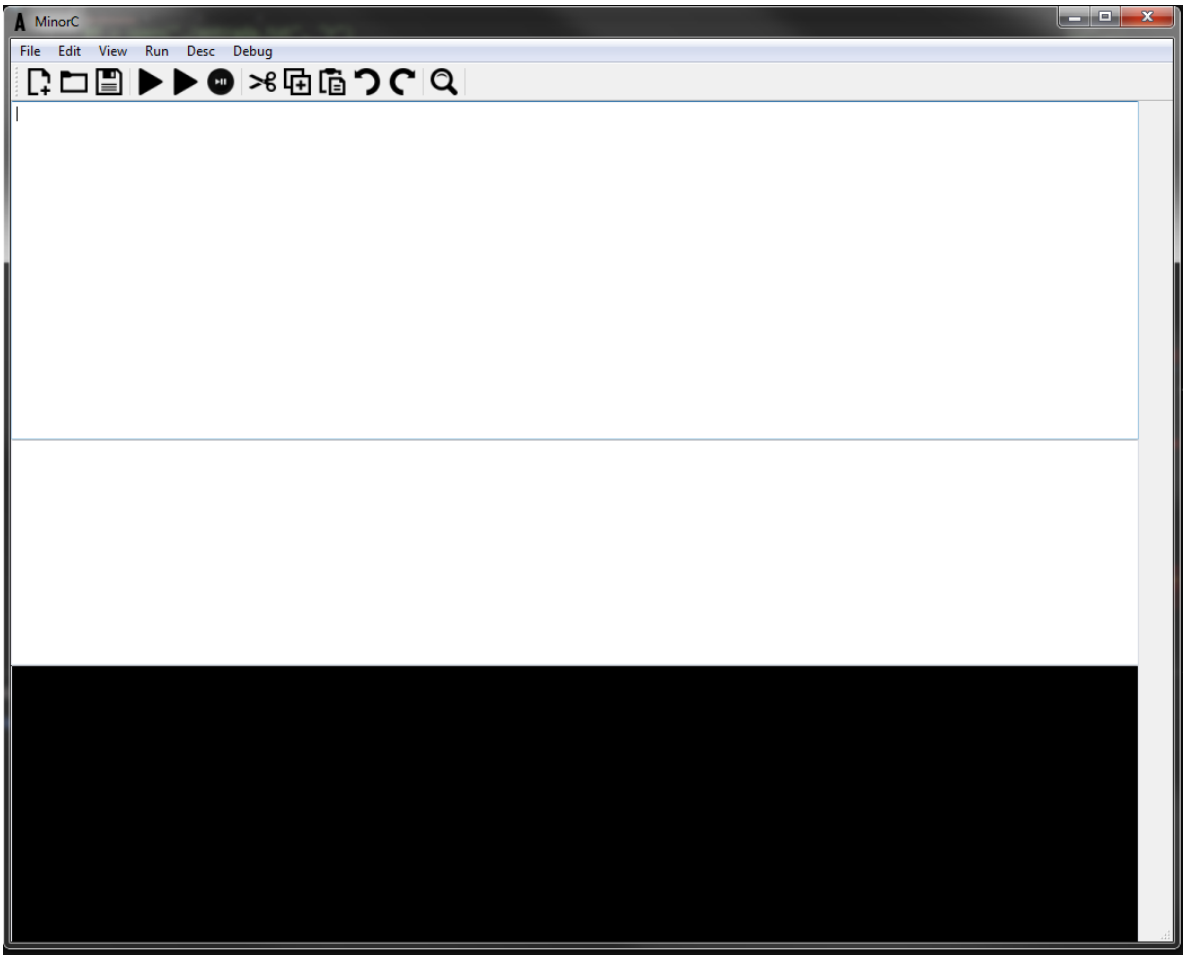
```
    error = "Error sintactico en el token \'" + str(t.value) +"\' en la line
a: "+ str(t.lineno) + ' columna:' + str(find_column(t.lexer.lexdata,t))
    lista_errores.append(error)


    return to
    #print(t)
    #print("Error sintáctico en '%s'" % t.value,'> ',str(t.lineno))
def getGramatical():
    return gramatical

import ply.yacc as yacc
parser = yacc.yacc()
#print("gramatical:",gramatical)

def parse(input) :

    return parser.parse(input)
```



En Archivo: Nuevo, Abrir, Guardar, Guardar Como, Cerrar y Salir.

- En Editar: las operaciones básicas de copiar, pegar, cortar, buscar, reemplazar, etc.
- En Ejecutar: ejecutar utilizando el analizador sintáctico ascendente o el descendente, abajo los tipos de reportes al ejecutar. Para la opción ascendente debe haber una opción ejecutar paso a paso para ver el estado de la pila (una forma de debbuging).
- En Opciones: cambiar color del fondo, quitar los números de línea, etc.
- En Ayuda: Ayuda y Acerca de.

Al ingresar código y ejecutarlo lo traduce a lenguaje Augus para luego ser interpretado por el interpete Augus

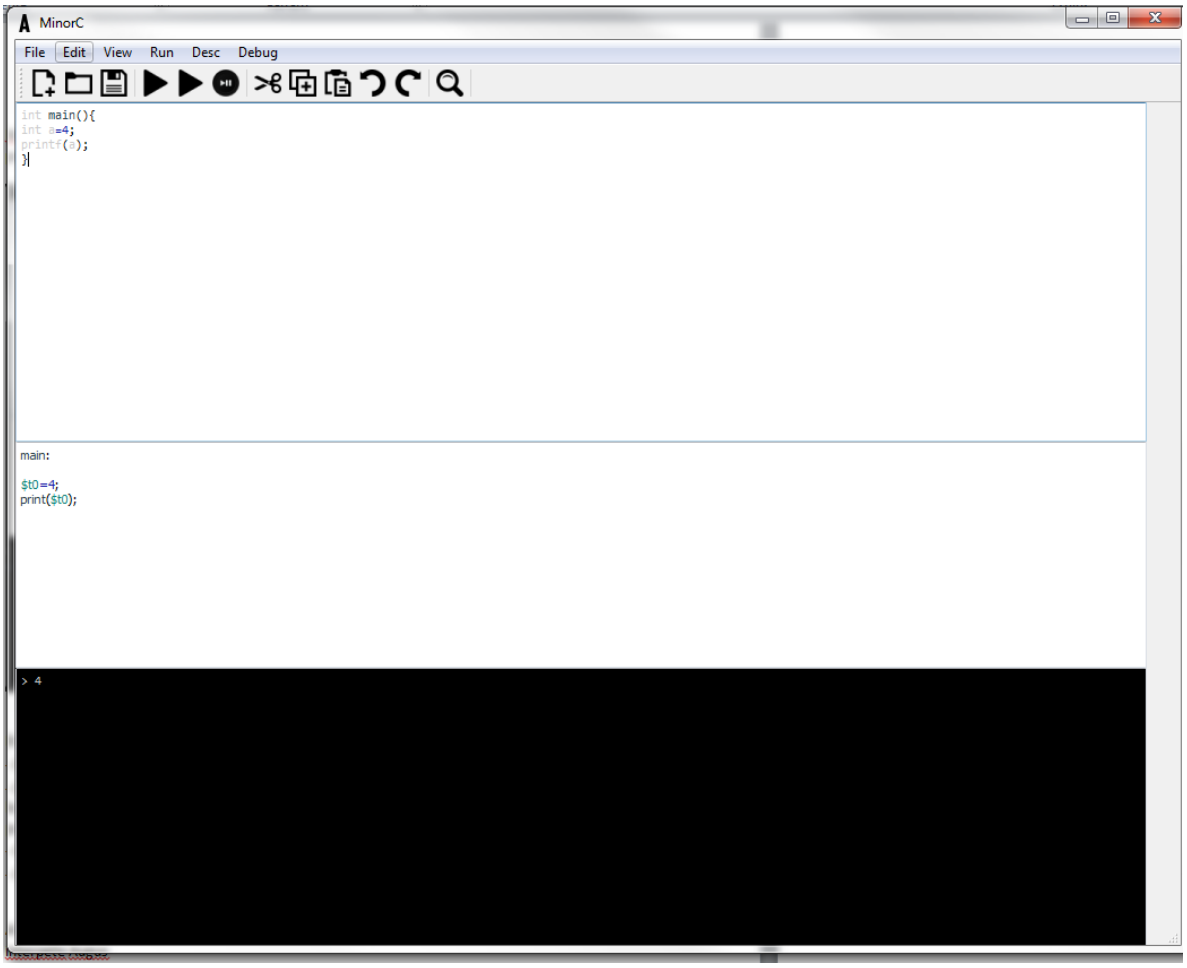


Tabla de símbolos MinorC

Gramatical MinorC

inicio.val : metodos.val
instrucciones.val: instruccion.val
instruccion.val: expresiones.val
metodo.val : id.val
instrucciones.val : instrucciones1.append(instruccion.val)
instruccion.val: expresiones.val
imprimir.val : expresion.val
expresion.val : id.val
instrucciones.val: instruccion.val
instruccion.val: expresiones.val
expresion.val : entero.val
lista_ids.val : expresion.val
expresion.val : id.val

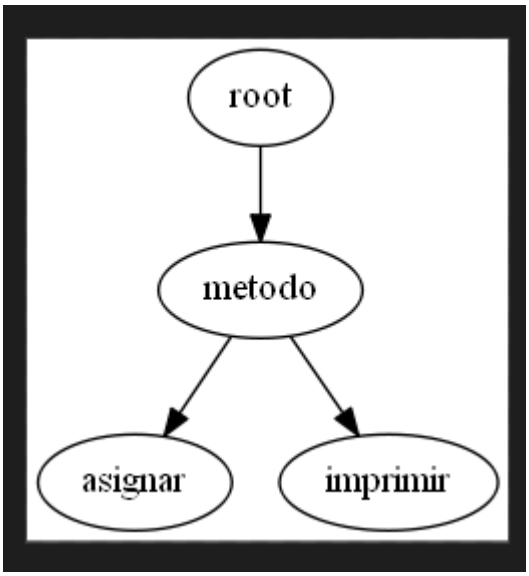
Tabla de Simbolos Augus

$t0	TIPO_DATO.NUMERO	4

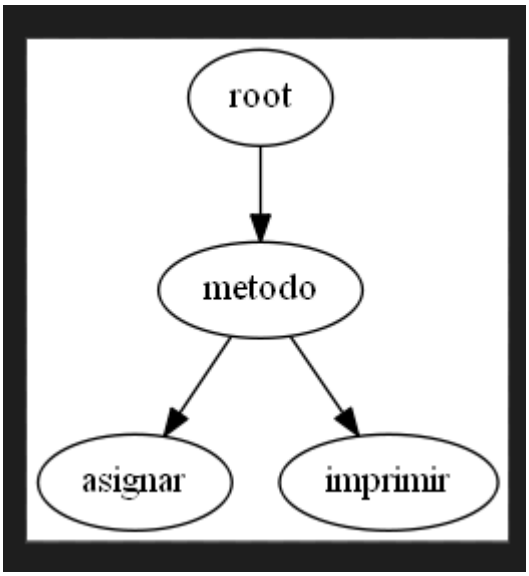
Gramatical Augus

inicio.val : instrucciones.val
instrucciones.val : instrucciones1.append(instruccion.val)
instruccion.val: expresiones.val
imprimir.val : expresion.val
expresion.val : temp.val
instrucciones.val : instrucciones1.append(instruccion.val)
instruccion.val: expresiones.val
asignartemp.val : expresion.val
expresion.val : entero.val
instrucciones.val: instruccion.val
instruccion.val: expresiones.val
metodo.val : id.val

Arbol MinorC


root
metodo
asignar
imprimir

Arbol Augus


root
metodo
asignar
imprimir