

DOCUMENTACION DE GRAMATICA MINOR C



Universidad de San Carlos de Guatemala
Juan Pablo Osuna de Leon
20153911




El metodo inicial de la gramatica se llamara Parser, en el cual se haran los import de las herramientas “**Lexer**” y “**Parser**”, los cuales seran instancias de PLY.

```
def parse(input):
    global input_, sintacticErroList, LexicalErrosList

    | sintacticErroList[:] = []
    LexicalErrosList[:] =[]

    input_ = input

    import ply.lex as lex
    import ply.yacc as yacc
    lexer = lex.lex()
    parser = yacc.yacc()
    instructions = parser.parse(input)
    print(str(instructions))
    lexer.lineno = 1
    parser.restart()
    if len(LexicalErrosList) > 0 or len(sintacticErroList) > 0: ...
    return instructions
```



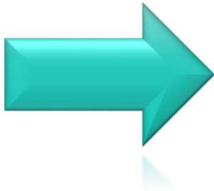
Analizador Lexico:

En esta sección definiremos los tokens que usaremos, así como las expresiones regulares, palabras reservadas etc.

Palabras Reservadas: son aquellas palabras que nuestro lenguaje determinara como reservadas, por ende, no pueden ser usadas como id u otra expresión.

```
/ reservadas = {
    'int': 'INT',
    'float': 'FLOAT',
    'char': 'CHAR',
    'double': 'DOUBLE',
    'printf': 'PRINTF',
    'struct': 'STRUCT',
    'break': 'BREAK',
    'case': 'CASE',
    'continue': 'CONTINUE',
    'default': 'DEFAULT',
    'if': 'IF',
    'else': 'ELSE',
    'for': 'FOR',
    'while': 'WHILE',
    'do': 'DO',
    'goto': 'GOTO',
    'return': 'RETURN',
    'sizeof': 'SIZEOF',
    'switch': 'SWITCH',
    'void': 'VOID',
    'scanf': 'SCANF',
    'malloc': 'MALLOC'
}
```

Tokens: son aquellos tokens que se reconocerán en el lenguaje, entiéndase símbolos, primero se realiza la declaración de tokens y posteriormente se les asigna el valor deseado, adjuntando al final nuestras palabras reservadas:

<pre>tokens = ['C_INT', 'C_CHAR', 'C_FLOAT', 'MASIGUAL', 'MENOSIGUAL', 'PORIGUAL', 'DIVIGUAL', 'MODIGUAL', 'SIIGUAL', 'SDIGUAL', 'ANDIGUAL', 'XORIGUAL', 'ORIGUAL', 'CHAR_',] + list(reservadas.values())</pre>		<pre># er tokens t_C_INT = r'\(int\)' t_C_CHAR = r'\(char\)' t_C_FLOAT = r'\(float\)' t_MASIGUAL = r'\+= ' t_MENOSIGUAL = r'\-= ' t_PORIGUAL = r'*=' t_DIVIGUAL = r'\/= ' t_MODIGUAL = r'\%=' t_SIIGUAL = r'\<\<=' t_SDIGUAL = r'\>\>=' t_ANDIGUAL = r'\&=' t_XORIGUAL = r'\^=' t_ORIGUAL = r'\ ='</pre>
---	---	--

Definición de expresiones regulares, estas expresiones serán usadas para definir un patrón de caracteres.

```
def t_NUMERO(t):
    r'\d+(\.\d+)?'
    try: ...
    except ValueError: ...
    return t

def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    #print(str(t.value))
    t.type = reservadas.get(t.value.lower(), 'ID')
    return t

def t_CHAR_(t): ...

def t_CADENA(t): ...

def t_COMENTARIO(t): ...
```

Se ignorarán definirán los caracteres que deseemos ignorar:

```
#ignored characters  
t_ignore = " \t"
```

Definimos la producción de error, el cual solo daremos un skip para que siga analizando todos los caracteres no permitidos y posteriormente poderlos reportar.

```
def t_error(t):  
    global input_, LexicalErrosList  
    lo = lexObj(t.value[0], find_column(input_, t), t.lexer.lineno)  
    LexicalErrosList.append(lo)  
    print("Illegal character '%s'" % t.value[0]+", linea: "+str(t.lexer.lineno))  
    t.lexer.skip(1)
```

Analizador Sintáctico:

Para el analizador sintáctico haremos uso de nuestra gramática.

Inicial definimos la precedencia, para este lenguaje se usará la precedencia de operadores de C, como se muestra en la siguiente imagen:

Precedencia y asociatividad

Nivel	Operadores	Descripción	Asoci.
1	() [] -> .	Acceso a un elemento de un vector y paréntesis	Izquierdas
2	+ - ! ~ * & ++ -- (cast) sizeof	Signo (unario), negación lógica, negación bit a bit Acceso a un elemento (unarios): puntero y dirección Incremento y decremento (pre y post) Conversión de tipo (<i>casting</i>) y tamaño de un elemento	Derechas
3	* / %	Producto, división, módulo (resto)	Izquierdas
4	+ -	Suma y resta	Izquierdas
5	>> <<	Desplazamientos	Izquierdas
6	< <= >= >	Comparaciones de superioridad e inferioridad	Izquierdas
7	== !=	Comparaciones de igualdad	Izquierdas
8	&	Y (<i>And</i>) bit a bit (binario)	Izquierdas
9	^	O-exclusivo (<i>Exclusive-Or</i>) (binario)	Izquierdas
10		O (<i>Or</i>) bit a bit (binario)	Izquierdas
11	&&	Y (<i>And</i>) lógico	Izquierdas
12		O (<i>Or</i>) lógico	Izquierdas
13	?:	Condición	Derechas
14	= *= /= %= += -= >>= <<= &= ^= =	Asignaciones	Derechas
15	,	Coma	Izquierdas

Una vez obtenida la precedencia del lenguaje C, se procede a definirla en nuestro analizador sintáctico.

```
precedence = (  
    ('left', 'COMA'),  
    ('right', 'IGUAL', 'MASIGUAL', 'MENOSIGUAL', 'PORIGUAL', 'DIVIGUAL', 'MODIGUAL', 'SIIGUAL', 'SDIGUAL', 'AIGUAL'),  
    ('right', 'TERNARIO'),  
    ('left', 'OR'),  
    ('left', 'AND'),  
    ('left', 'ORBIT'),  
    ('left', 'XORBIT'),  
    ('left', 'ANDBIT'),  
    ('left', 'DIFERENTE', 'IGUALQUE'),  
    ('left', 'MENORQUE', 'MAYORQUE', 'MENORIGUAL', 'MAYORIGUAL'),  
    ('left', 'SHIFTIZQ', 'SHIFTDER'),  
    ('left', 'MAS', 'MENOS'),  
    ('left', 'POR', 'DIV', 'MODULO'),  
    ('right', 'C_INT', 'C_FLOAT', 'C_CHAR', 'INCREMENTO', 'DECREMENTO', 'UMENOS', 'UANDBIT', 'NOTLOGICO'),  
    ('left', 'PARIZQ', 'CORIZQ')  
)
```

Una vez definida nuestra precedencia de operadores definimos una producción de inicio:

(grammarList servirá para concatenar todas las producciones para posterior uso de reporte gramatical)

Declaración de lista grammarList:

```
grammarList = []  
grammarList[:] = []
```

Producción de Inicio:

```
#definition of grammar  
✓ def p_init(t):  
    'S : A'  
    t[0] = t[1]  
    print("Se ha reconocido la cadena.")  
    global grammarList  
    grammarList.append(g.nodeGramatical('S -> A', f'S.val = A.val'))  
    grammarList.reverse()
```

Nuestra gramática contendrá una lista de instrucciones globales que será la principal y la encargada de concatenar todas las instrucciones, por medio del método “append” de una lista y al final solo se sintetiza el valor.

```
def p_listaInstrucciones(t):
    '''INSTRUCCIONES_GLOBALES : INSTRUCCIONES_GLOBALES DECLARACION_GLOBAL
    | DECLARACION_GLOBAL'''

    global grammarList
    if len(t) == 3:
        t[1].append(t[2])
        t[0] = t[1]
        grammarList.append(g.nodeGramatical('INSTRUCCIONES_GLOBALES -> INSTRUCCIONES_GLOBALES
    else:
        t[0] = [t[1]]
        grammarList.append(g.nodeGramatical('INSTRUCCIONES_GLOBALES -> DECLARACION_GLOBAL', f'
```

Una declaración global solo tendrá 3 posibles instrucciones, las cuales serán, las declaraciones de variables y arreglos globales, declaración de structs, y declaración de métodos, la instrucción solo se sintetiza por medio de “t[0] = t[1]”.

```
def p_declaracionGlobal(t):
    '''DECLARACION_GLOBAL : DECLA_VARIABLES
    | DECLA_FUNCIONES
    | DECLA_STRUCTS'''

    t[0] = t[1]
```

Una declaración de variable esta conformada por un tipo, una lista de Ids y al finalizar un punto y coma.

```
def p_declaVariable(t):
    'DECLA_VARIABLES : TIPO LISTA_ID PUNTOCOMA'
    #print(f'tipo: {str(t[1])} valor: {str(t[2])}')
    t[0] = Declaration(t[1], t.lineno(1), t.lexpos(1), t[2])
    global grammarList
    grammarList.append(g.nodeGramatical('DECLA_VARIABLES -> TI
```

La lista de Id, será una lista de Ids con posible asignación separados por coma.

```
def p_listaId(t):
    '''LISTA_ID :    LISTA_ID COMA ASIGNA
    |            | ASIGNA'''

    global grammarList
    if len(t) == 2:
        t[0] = [t[1]]
        grammarList.append(g.nodeGramatical('LI
    else:
        t[1].append(t[3])
        t[0] = t[1]
        grammarList.append(g.nodeGramatical('LI
```

La asignación servirá como ejemplo para el uso de nuestra clase Abstracta Instrucción:

```
def p_asigna(t):
    '''ASIGNA :    ID IGUAL EXPRESION
    |            | ID CORCHETES
    |            | ID CORCHETES IGUAL EXPRESION
    |            | ID'''

    global grammarList
    if len(t) == 4:
        t[0] = SingleDeclaration(t[1], t[3], t.lineno(2))
        grammarList.append(g.nodeGramatical('ASIGNA -> :
    elif len(t) == 3:
        t[0] = IdentifierArray(t[1], t[2], t.lineno(1), t
        grammarList.append(g.nodeGramatical('ASIGNA -> :
    elif len(t) == 2:
        t[0] = SingleDeclaration(t[1], '#', t.lineno(1),
        grammarList.append(g.nodeGramatical('ASIGNA -> :
    else: ...
```

Uso de la precedencia en la gramática:

Servirá la precedencia ya que utilizamos una gramática ambigua, siendo el caso de una expresión, esto servirá para que el analizador sepa por donde desplazar y por reducir .

```
def p_expression(t):
    '''EXPRESSION : EXPRESSION MAS EXPRESSION
    | EXPRESSION MENOS EXPRESSION
    | EXPRESSION POR EXPRESSION
    | EXPRESSION DIV EXPRESSION
    | EXPRESSION MODULO EXPRESSION
    | EXPRESSION IGUALQUE EXPRESSION
    | EXPRESSION DIFERENTE EXPRESSION
    | EXPRESSION MENORQUE EXPRESSION
    | EXPRESSION MAYORQUE EXPRESSION
    | EXPRESSION MENORIGUAL EXPRESSION
    | EXPRESSION MAYORIGUAL EXPRESSION
    | EXPRESSION NOTBIT EXPRESSION
    | EXPRESSION ANDBIT EXPRESSION
    | EXPRESSION XORBIT EXPRESSION
    | EXPRESSION ORBIT EXPRESSION
    | EXPRESSION OR EXPRESSION
    | EXPRESSION AND EXPRESSION
    | EXPRESSION SHIFTIZQ EXPRESSION
    | EXPRESSION SHIFTER EXPRESSION
    | PARIZQ EXPRESSION PARDER
    | C_INT EXPRESSION
    | C_FLOAT EXPRESSION
    | C_CHAR EXPRESSION
    | SIZEOF PARIZQ EXPRESSION PARDER
    | NOTLOGICA EXPRESSION
    | MENOS EXPRESSION %prec Umenos
    | NOTBIT EXPRESSION
    | ANDBIT EXPRESSION %prec UANDBIT
    | LLAMADA_FUNCION
    | SCANF PARIZQ PARDER
    | INCRE_DECRE'''
    # EXPRESSION TERNARIO EXPRESSION DOSPUNTOS EXPRESSION'''

global grammarList
if len(t) == 4:
    #aritméticos
    if t[2] == '+': t[0] = BinaryExpression(t[1],t[3],Aritméticos.MAS, t.lineno(2), t.lexpos(2))
```


Producción de Error:

La recuperación de errores es una parte muy importante de todo programa, ya que no queremos que, al encontrar el primer error, se detenga la ejecución, si no que queremos que se puedan reportar la mayoría de los errores al usuario; para ello utilizamos una producción de error, cuya función es analizar el error y desplazar hasta que encuentre un token de resincronización, en este caso punto y coma.

```
##-----DECLARACION DE V
def p_declaVariable_error(t):
    'DECLA_VARIABLES : TIPO error PUNTOCOMA'
def p_declaVariable(t):
    'DECLA_VARIABLES : TIPO LISTA_ID PUNTOCOMA'
```

La producción de error es una copia de la producción normal, a diferencia que los tokens que pudieran contener un error los omito con la palabra “error” y se sincroniza con “PUNTOCOMA”;

Aparte tenemos nuestra producción de error donde el error ya no se puede recuperar.

```
def p_error(t):
    global sintacticErroList
    try:
        print("Error sintactico en '%s'" % t.value + "line: " + str(t.lineno))
        so = sinOb(t.value, t.lineno, find_column(input_, t))
        sintacticErroList.append(so)
    except:
        print("Error sintactico Irrecuperable")

        so = sinOb('Error sintactico: Irrecuperable', 0, 0)
        sintacticErroList.append(so)
```

Anexos Gramática completa:

Repositorio:

- https://drive.google.com/file/d/14fhVAJ5rp8O0nFEENYikYVH74_S0LcCx/view?usp=sharing

Análisis Léxico:

```
reservadas = {  
    'int': 'INT',  
    'float': 'FLOAT',  
    'char': 'CHAR',  
    'double': 'DOUBLE',  
    'printf': 'PRINTF',  
    'struct': 'STRUCT',  
    'break': 'BREAK',  
    'case': 'CASE',  
    'continue': 'CONTINUE',  
    'default': 'DEFAULT',  
    'if': 'IF',  
    'else': 'ELSE',  
    'for': 'FOR',  
    'while': 'WHILE',  
    'do': 'DO',  
    'goto': 'GOTO',  
    'return': 'RETURN',  
    'sizeof': 'SIZEOF',  
    'switch': 'SWITCH',  
    'void': 'VOID',  
    'scanf': 'SCANF',  
    'malloc': 'MALLOC'  
}
```

```
tokens = [  
    'C_INT',  
    'C_CHAR',  
    'C_FLOAT',  
    'MASIGUAL',  
    'MENOSIGUAL',  
    'PORIGUAL',  
    'DIVIGUAL',  
    'MODIGUAL',  
    'SIIGUAL',  
    'SDIGUAL',  
    'ANDIGUAL',  
    'XORIGUAL',  
    'ORIGUAL',  
    'INCREMENTO',  
    'DECREMENTO',  
    'PUNTO',  
    'FLECHA',  
    'TERNARIO',  
]
```

```
'COMA',  
'PUNTOCOMA',  
'DOSPUNTOS',  
'LLAVEIZQ',  
'LLAVEDER',  
'PARIZQ',  
'PARDER',  
'CORIZQ',  
'CORDER',  
'NOTBIT',  
'ANDBIT',  
'ORBIT',  
'XORBIT',  
'SHIFTIZQ',  
'SHIFTDER',  
'NOTLOGICA',  
'MENOS',  
'MAS',  
'POR',  
'DIV',  
'MODULO',  
'AND',  
'OR',  
'IGUAL',  
'IGUALQUE',  
'DIFERENTE',  
'MAYORIGUAL',  
'MENORIGUAL',  
'MAYORQUE',  
'MENORQUE',  
'NUMERO',  
'ID',  
'CADENA',  
'CHAR_'  
] + list(reservadas.values())
```

```

# er tokens
t_C_INT      = r'\(int\,'
t_C_CHAR     = r'\(char'
t_C_FLOAT    = r'\(floa
t_MASIGUAL  = r'\+= '
t_MENOSIGUAL = r'\-= '
t_PORIGUAL  = r'\*= '
t_DIVIGUAL  = r'\/= '
t_MODIGUAL  = r'\%='
t_SIIGUAL   = r'\<<='
t_SDIGUAL   = r'\>>='
t_ANDIGUAL  = r'\&='
t_XORIGUAL  = r'\^='
t_ORIGUAL   = r'\|='
t_INCREMENTO = r'\++ '
t_DECREMENTO = r'\-- '
t_PUNTO     = r'\.'
t_FLECHA    = r'\->'
t_TERNARIO  = r'\?'
t_COMA      = r'\,'
t_PUNTOCOMA = r'\;'
t_DOSPUNTOS = r'\:'
t_LLAVEIZQ  = r'\{'
t_LLAVEDER  = r'\}'
t_PARIZQ    = r'\('
t_PARDER    = r'\)'
t_CORIZQ    = r'\['
t_CORDER    = r'\]'
t_NOTBIT    = r'\~'
t_ANDBIT    = r'\&'
t_ORBIT     = r'\|'
t_XORBIT    = r'\^'
t_SHIFTIZQ  = r'\<<'
t_SHIFTER   = r'\>>'
t_NOTLOGICA = r'\!'
t_MENOS     = r'\-'
t_MAS       = r'\+'
t_POR       = r'\*'
t_DIV       = r'\/'
t_MODULO    = r'\%'
t_AND       = r'\&\&'
t_OR       = r'\|\|'
t_IGUAL     = r'\='
t_IGUALQUE  = r'\== '
t_DIFERENTE = r'\!= '
t_MAYORIGUAL = r'\>='
t_MENORIGUAL = r'\<='
t_MAYORQUE  = r'\>'
t_MENORQUE  = r'\<'

```