

Diseño e Implementación de un Analizador
Léxico y Analizador Sintáctico para el
Lenguaje AVISMO

Aramis E. Matos

Lenier Gerena

Angel Berrios Pellot

Segundo Semestre, 2022-2023

Tabla de Contenido

1	Introducción	3
2	Analizador Léxico	5
2.1	Gramatica del Lenguaje AVISMO	5
2.2	Diseño del del Analizador Léxico	13
2.2.1	Autómatas Finitos Deterministas	13
2.2.2	Tabla de Símbolos	14
2.3	Implementación del Analizador Léxico	15
3	Analizador Sintáctico	20
3.1	<i>grammar.py</i>	20
3.2	<i>tester.py</i>	23
4	Conclusiones y Recomendaciones	24
A	Código de Analizador Léxico	26
B	Código de Analizador Sintáctico	36

C Casos de Prueba	44
Referencias Bibliográficas	54

Capítulo 1

Introducción

La visualización molecular puede ser considerada como una de las áreas mas importante dentro de la bioinformática. Entre sus aplicaciones mas relevantes se destacan el diseño de nuevo fármacos... (Narciso Farias, Rios, Hidrobo, & Vicuña, 2012). Este enunciado fue escrito hace mas de una década. Sin embargo, hoy día en un mundo pospandemia, reconocemos que tan sabio fue. El desarrollo de la vacuna contra el COVID-19 tan rápido fue gracias a herramientas de visualización como el Ambiente de visualización Molecular (AVISMO) (Narciso Farias et al., 2012) El propósito de este proyecto es definir el automata de estado finito del lenguaje AVISMO, los patrones el cual caracterizan los lexemas del lenguaje, los atributos de los lexemas y la implementación del analizador léxico y sintáctico del lenguaje AVISO en Python. La implementación léxica y sintáctica fue desarrollada utilizando Python Lex Yacc (*PLY (Python Lex-Yacc) — Ply 4.0 Documentation*, n.d.) con el lenguaje MAPL (*PL-Project-LGM-YVV-AMN/MAPL*, n.d.)

como base.

Capítulo 2

Analizador Léxico

2.1 Gramatica del Lenguaje AVISMO

- $\langle \text{SENTENCIAS} \rangle ::= \langle \text{FIN_DE_LINEA} \rangle \langle \text{SENTENCIAS} \rangle \mid \langle \text{SENTENCIA} \rangle \langle \text{FIN_DE_LINEA} \rangle$
- $\langle \text{FIN_DE_LINEA} \rangle ::= ":" \mid ";;"$
- $\langle \text{SENTENCIA} \rangle ::= \text{"defina"} \langle \text{ID} \rangle \text{"como"} \langle \text{TIPO} \rangle \mid \langle \text{ID} \rangle \text{"="} \langle \text{MODELO_MOLECULAR} \rangle \mid \langle \text{OPERACION} \rangle \text{"("} \langle \text{ID} \rangle \text{"}"}$
- $\langle \text{ID} \rangle ::= \text{"A"} \mid \text{"B"} \mid \text{"C"} \mid \text{"D"} \mid \text{"E"} \mid \text{"F"} \mid \text{"G"} \mid \text{"H"} \mid \text{"I"} \mid \text{"J"} \mid \text{"K"} \mid \text{"L"} \mid \text{"M"} \mid \text{"N"} \mid \text{"O"} \mid \text{"P"} \mid \text{"Q"} \mid \text{"R"} \mid \text{"S"} \mid \text{"T"} \mid \text{"U"} \mid \text{"V"} \mid \text{"W"} \mid \text{"X"} \mid \text{"Y"} \mid \text{"Z"} \mid \text{"a"} \mid \text{"b"} \mid \text{"c"} \mid \text{"d"} \mid \text{"e"} \mid \text{"f"} \mid \text{"g"} \mid \text{"h"} \mid \text{"i"} \mid \text{"j"} \mid \text{"k"} \mid \text{"l"} \mid \text{"m"} \mid \text{"n"} \mid \text{"o"} \mid \text{"p"} \mid \text{"q"} \mid \text{"r"} \mid \text{"s"} \mid \text{"t"} \mid \text{"u"} \mid \text{"v"} \mid \text{"w"} \mid \text{"x"} \mid \text{"y"} \mid \text{"z"} \mid \langle \text{LETRA} \rangle \langle \text{IDCONT} \rangle$

- <IDCONT> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | <LETRA> <IDCONT> | "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" | <DIGITO> <IDCONT>
- <LETRA> ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
- <DIGITO> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
- <TIPO> ::= "modelo"
- <OPERACION> ::= "graficar2d" | "graficar3d" | "pesomolecular"
- <MODELO_MOLECULAR> ::= "H" | "Li" | "Na" | "K" | "Rb" | "Cs" | "Fr" | "Be" | "Mg" | "Ca" | "Sr" | "Ba" | "Ra" | "Sc" | "Y" | "Ti" | "Zr" | "Hf" | "Db" | "V" | "Nb" | "Ta" | "Ji" | "Cr" | "Mo" | "W" | "Rf" | "Mn" | "Tc" | "Re" | "Bh" | "Fe" | "Ru" | "Os" | "Hn" | "Co" | "Rh" | "Ir" | "Mt" | "Ni" | "Pd" | "Pt" | "Cu" | "Ag" | "Au" | "Zn" | "Cd" | "Hg" | "B" | "Al" | "Ga" | "In" | "Ti" | "C" | "Si" | "Ge" | "Sn" | "Pb" | "N" | "P" | "As" | "Sb" | "Bi" | "O" | "S" | "Se" | "Te" | "Po" | "F" | "Cr" | "Br" | "I" | "At" | "He" | "Ne" | "Ar" | "Kr" | "Xe" | "Rn" | <ELEMENTO_QUIMICO> <VALENCIA> | <ELE-

MENTO> <GRUPO_FUNCIONAL> | <COMPUESTO> <ELEMENTO>
 <GRUPO_FUNCIONAL> | <COMPUESTO> <COMPUESTO>

- <COMPUESTO> ::= "H" | "Li" | "Na" | "K" | "Rb" | "Cs" | "Fr" | "Be" | "Mg" | "Ca" | "Sr" | "Ba" | "Ra" | "Sc" | "Y" | "Ti" | "Zr" | "Hf" | "Db" | "V" | "Nb" | "Ta" | "Ji" | "Cr" | "Mo" | "W" | "Rf" | "Mn" | "Tc" | "Re" | "Bh" | "Fe" | "Ru" | "Os" | "Hn" | "Co" | "Rh" | "Ir" | "Mt" | "Ni" | "Pd" | "Pt" | "Cu" | "Ag" | "Au" | "Zn" | "Cd" | "Hg" | "B" | "Al" | "Ga" | "In" | "Ti" | "C" | "Si" | "Ge" | "Sn" | "Pb" | "N" | "P" | "As" | "Sb" | "Bi" | "O" | "S" | "Se" | "Te" | "Po" | "F" | "Cr" | "Br" | "I" | "At" | "He" | "Ne" | "Ar" | "Kr" | "Xe" | "Rn" | <ELEMENTO_QUIMICO> <VALENCIA> | <ELEMENTO> <GRUPO_FUNCIONAL> | <ELEMENTO> <GRUPO_FUNCIONAL> <ENLACE> | <ELEMENTO> <ENLACE>
- <COMPUESTOS> ::= <COMPUESTO> <COMPUESTO> | <COMPUESTOS>
- <ELEMENTO> ::= "H" | "Li" | "Na" | "K" | "Rb" | "Cs" | "Fr" | "Be" | "Mg" | "Ca" | "Sr" | "Ba" | "Ra" | "Sc" | "Y" | "Ti" | "Zr" | "Hf" | "Db" | "V" | "Nb" | "Ta" | "Ji" | "Cr" | "Mo" | "W" | "Rf" | "Mn" | "Tc" | "Re" | "Bh" | "Fe" | "Ru" | "Os" | "Hn" | "Co" | "Rh" | "Ir" | "Mt" | "Ni" | "Pd" | "Pt" | "Cu" | "Ag" | "Au" | "Zn" | "Cd" | "Hg" | "B" | "Al" | "Ga" | "In" | "Ti" | "C" | "Si" | "Ge" | "Sn" | "Pb" | "N" | "P" | "As" | "Sb" | "Bi" | "O" | "S" | "Se" | "Te" | "Po" | "F" | "Cr" | "Br" | "I" | "At" | "He" | "Ne" | "Ar" | "Kr" | "Xe" | "Rn" | <ELEMENTO_QUIMICO> <VALENCIA>
- <ELEMENTO_QUIMICO> ::= "H" | "Li" | "Na" | "K" | "Rb" | "Cs" | "Fr" |

"Be" | "Mg" | "Ca" | "Sr" | "Ba" | "Ra" | "Sc" | "Y" | "Ti" | "Zr" | "Hf" | "Db" |
 "V" | "Nb" | "Ta" | "Ji" | "Cr" | "Mo" | "W" | "Rf" | "Mn" | "Tc" | "Re" | "Bh" |
 "Fe" | "Ru" | "Os" | "Hn" | "Co" | "Rh" | "Ir" | "Mt" | "Ni" | "Pd" | "Pt" | "Cu"
 | "Ag" | "Au" | "Zn" | "Cd" | "Hg" | "B" | "Al" | "Ga" | "In" | "Ti" | "C" | "Si"
 | "Ge" | "Sn" | "Pb" | "N" | "P" | "As" | "Sb" | "Bi" | "O" | "S" | "Se" | "Te" |
 "Po" | "F" | "Cr" | "Br" | "I" | "At" | "He" | "Ne" | "Ar" | "Kr" | "Xe" | "Rn"

- <VALENCIA> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
- <GRUPO_FUNCIONAL> ::= <GRUPO_FUNCIONAL_INFERIOR>
 <GRUPO_FUNCIONAL_SUPERIOR> | <GRUPO_FUNCIONAL_SUPERIOR>
 <GRUPO_FUNCIONAL_INFERIOR> | "(" <MODELO_GRUPO_FUNCIONAL>
 ")" | "[" <MODELO_GRUPO_FUNCIONAL> "]"
- <GRUPO_FUNCIONAL_SUPERIOR> ::= "[" <MODELO_GRUPO_FUNCIONAL>
 "]"
- <GRUPO_FUNCIONAL_INFERIOR> ::= "(" <MODELO_GRUPO_FUNCIONAL>
 ")"
- <MODELO_GRUPO_FUNCIONAL> ::= <ENLACE> <MODELO_MOLECULAR>
 | "H" | "Li" | "Na" | "K" | "Rb" | "Cs" | "Fr" | "Be" | "Mg" | "Ca" | "Sr" | "Ba"
 | "Ra" | "Sc" | "Y" | "Ti" | "Zr" | "Hf" | "Db" | "V" | "Nb" | "Ta" | "Ji" | "Cr"
 | "Mo" | "W" | "Rf" | "Mn" | "Tc" | "Re" | "Bh" | "Fe" | "Ru" | "Os" | "Hn" |
 "Co" | "Rh" | "Ir" | "Mt" | "Ni" | "Pd" | "Pt" | "Cu" | "Ag" | "Au" | "Zn" | "Cd" |
 "Hg" | "B" | "Al" | "Ga" | "In" | "Ti" | "C" | "Si" | "Ge" | "Sn" | "Pb" | "N" | "P"

| "As" | "Sb" | "Bi" | "O" | "S" | "Se" | "Te" | "Po" | "F" | "Cr" | "Br" | "I" | "At"
 | "He" | "Ne" | "Ar" | "Kr" | "Xe" | "Rn" | <ELEMENTO_QUIMICO> <VA-
 LENCIA> | <ELEMENTO> <GRUPO_FUNCIONAL> | <COMPUESTO>
 <ELEMENTO> | <COMPUESTO> <COMPUESTO> <COMPUESTOS>

En la tabla 2.1, en la columna de patrones, note que cuando dice $\{TOKEN\}$ donde *TOKEN* se refiere a el patrón asociado a *token*. Por ejemplo, si un patrón dice $\{ELEMENTO_QUIMICO\}$, esto significa que inserta el patrón asociado al *token ELEMENTO_QUIMICO*. Esto no significa que el analizador léxico espera un *token* de por si, sencillamente se hizo con el propósito de evitar redundancias.

<i>Token</i>	Patrón	Lexema	Atributos
<FIN_DE_LINEA>	; :	:	Símbolo reservado
<PALABRA _RESERVADA>	defina como	defina	Palabra reservada
<ID>	[A-Za-z][A-Za-z0-9]*	var1	Modelo molecular asociado
<IDCONT>	[A-Za-z0-9]+	1ar	ID asociado
<LETRA>	[A-Za-z]	a	ID asociado
<DIGITO>	[0-9]	7	Valor numérico, lexema asociado
<TIPO>	modelo	modelo	ID asociado
<OPERACION>	graficar2d graficar3d pesomolecular	pesomolecular	ID asociado
<MODELO _MOLECULAR>	({ELEMENTO _QUIMICO} {ELEMENTO _QUIMICO} {VALENCIA} {ELEMENTO} {GRUPO _FUNCIONAL} {ELEMENTO} {GRUPO _FUNCIONAL} {ENLACE} {ELEMENTO} {ENLACE})	CH3(CH3)CHH	ID asociado

<COMPUESTO>	COMPUESTO ({ELEMENTO _QUIMICO} {ELEMENTO _QUIMICO} {VALENCIA}) {ELEMENTO} {GRUPO_FUNCIONAL} {ELEMENTO} {GRUPO_FUNCIONAL} {ENLACE} {ELEMENTO} {ENLACE})	CH3::	Modelo molecular asociado, enlaces, valencias
<COMPUESTOS>	{COMPUESTO}+	CH3:::(OH)3	Modelo molecular asociado, enlaces, valencias
<ELEMENTO>	{ELEMENTO _QUIMICO} {VALENCIA}?	Ag3	Elemento, valencia
<ELEMENTO _QUIMICO>	("H" "Li" "Na" "K" "Rb" "Cs" "Fr" "Be" "Mg" "Ca" "Sr" "Ba" "Ra" "Sc" "Y" "Ti" "Zr" "Hf" "Db" "V" "Nb" "Ta" "Ji" "Cr" "Mo" "W" "Rf" "Mn" "Tc" "Re" "Bh" "Fe" "Ru" "Os" "Hn" "Co" "Rh" "Ir" "Mt" "Ni" "Pd" "Pt" "Cu" "Ag" "Au" "Zn" "Cd" "Hg" "B" "Al" "Ga" "In" "Tl" "C" "Si" "Ge" "Sn" "Pb" "N" "P" "As" "Sb" "Bi" "O" "S" "Se" "Te" "Po" "F" "Cr" "Br" "I" "At" "He" "Ne" "Ar" "Kr" "Xe" "Rn")	I	Elemento
<VALENCIA>	[1-9]	2	Valor

<GRUPO _FUNCIONAL>	({GRUPO _FUNCIONAL _INFERIOR} {GRUPO _FUNCIONAL _SUPERIOR} {GRUPO _FUNCIONAL _SUPERIOR} {GRUPO _FUNCIONAL _INFERIOR} "(" {MODELO _GRUPO _FUNCIONAL} ")" "[" MODELO _GRUPO _FUNCIONAL "]")	(CH3){Ag2}	Grupos funcionales, grupo funcional inferior, grupo funcional superior
<GRUPO _FUNCIONAL _INFERIOR>	"[" {MODELO _GRUPO _FUNCIONAL} "]"	[CVHe3]	Elementos, valencias
<GRUPO _FUNCIONAL _SUPERIOR>	"(" {MODELO _GRUPO _FUNCIONAL} ")"	(CVHe3)	Elementos, valencias
<MODELO _GRUPO _FUNCIONAL>	((ELEMENTO _QUIMICO)+ {VALENCIA}?) + ({ELEMENTO}+ {ENLACE} {ELEMENTO}+)+	FeH=C3Si4	Elementos, enlaces, valencias
<ENLACE>	("-" "=" ":" "::")	-	Valencia

Tabla 2.1: Tabla de Componentes Léxicos de AVISMO

2.2 Diseño del del Analizador Léxico

2.2.1 Autómatas Finitos Deterministas

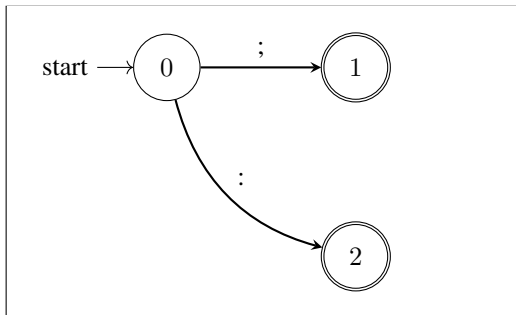


Figura 2.1: Automata del patrón para el token <FIN_DE_LINEA>

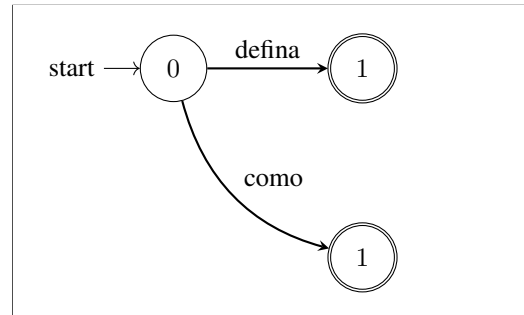


Figura 2.2: Automata del patrón para el token <PALABRAS_RESERVADA>

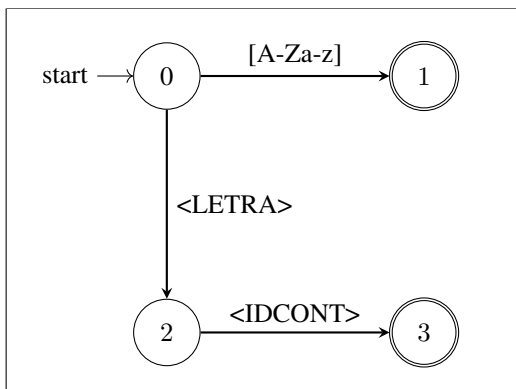


Figura 2.3: Automata del patrón para el token <ID>

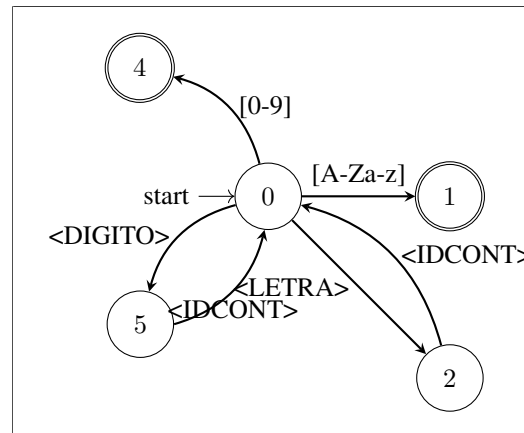


Figura 2.4: Automata del patrón para el token <IDCONT>

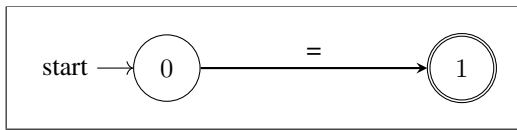


Figura 2.5: Automata del patrón para el token <ASIGNACION>

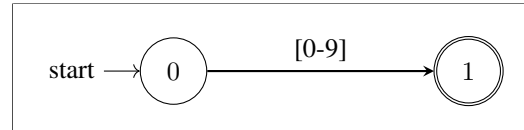


Figura 2.6: Automata del patrón para el token <LETRA>

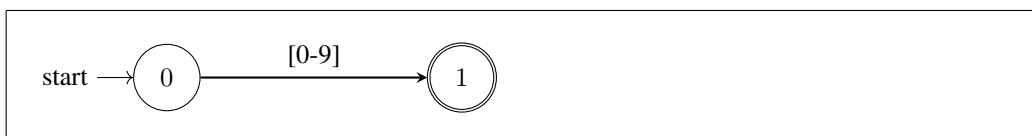


Figura 2.7: Automata del patrón para el token <DIGITO>

2.2.2 Tabla de Símbolos

Identificador

Atributo

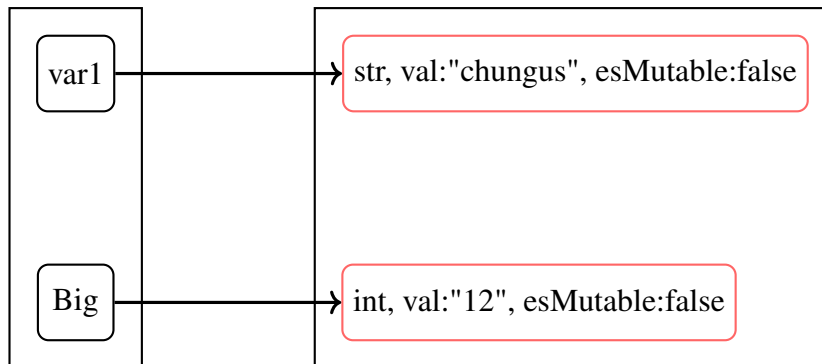


Figura 2.8: Tabla de símbolos implementada como un diccionario

2.3 Implementación del Analizador Léxico

Como se ha mencionado anteriormente, la implementación lexica del proyecto fue inspirada por la MAPL (*PL-Project-LGM-YVV-AMN/MAPL*, n.d.) y adaptada para la gramática de AVISMO.

Para ejecutar el analizador léxico, primero se tiene que instalar *Python 3*. Luego, se ejecuta `pip install ply` por la línea de comando. Finalmente, se ejecuta `python lexer.py test_prog.txt`

Al ejecutarse el comando anterior, el programa procede a leer **cada caracter** del programa e identificar si una serie de caracteres sigue un patrón que forma parte del lenguaje AVISMO. Al encontrar un patrón reconocido, tales como un identificador o modelo molecular, lo clasifica con un *token* correspondiente, lo emprime en el archivo de *output.txt* y lo devuelve al analizador sintáctico. Note, que el patrón de identificador reconoce palabra reservadas también. Esto crea ambigüedad semántica debido a que la gramática no tiene un mecanismo para diferenciar entre una palabra reservada y un identificador. Por esta razón, si una serie de caracteres se identifica como un lexema de categoría identificador, se compara con los valores ya existentes del diccionario *variables*. Al inicializar el programa `lexer.py`, este se encarga de abrir el archivo de palabras reservadas (*keywords.txt*) y añadir las palabras reservadas antes que cualquier variable se pueda inicializar. Mas aún, a las palabras reservadas se les asigna el valor de la cadena vacía. Esto se hace con el propósito de poder diferenciar entre palabras reservadas e identificadores, ya que al nivel sintáctico, no es posible asignarle a

una identificador una cadena vacía, como se puede apreciar a continuación:

```
def t_ID(t):                                #identifica el token ID
    ↪ pero tambien idetifica el token de palabra
    ↪ reservada
    r"[A-Za-z]+\d*"
    isPR = reserved.get(t.value, "ID")
    if isPR != "ID":
        t.type = "PALABRA_RESERVADA"
    else:
        t.type = isPR
        variables[t.value] = ""
    return t
```

Figura 2.9: Patrón que cezga identificadores de palabras reservadas

Con el propósito de visualizar los lexemas generados por `lexer.py`, colocado en el archivo *lexer.py*, se utiliza la siguiente función:

```

1  with open(test_file, "r") as f:
2      with open("output.txt", "w") as o:
3          for data in f:
4              lexer.input(data)
5              for tok in lexer:
6                  tokenTable.add_row([tokenNum, tok.type,
6                  ↪ , tok.value, tok.lineno, tok.lexpos,
6                  ↪ test_file])
7                  tokenNum += 1
8              o.write(str(tokenTable))
9              line = "\n\nTABLA DE SIMBOLOS"
10             o.write(line+"\n")
11             for val in variables:
12                 symbofigable.add_row([val])
13             o.write(str(symbofigable)+"\n")
14             line = "\n\nPALABRAS RESERVADAS"
15             o.write(line+"\n")
16             for val in reserved:
17                 reservedWords.add_row([val])
18             o.write(str(reservedWords)+"\n")

```

Figura 2.10: Código para imprimir los *tokens* encontrados

```
def t_error(t): # identifica error lexico
    tokenTable.addRow([tokenNum, "ERROR", t.value[0], t,
        ↪ .lineno, t.lexpos, test_file])
    t.lexer.skip(1)
```

Figura 2.11: Gestión de errores

Esto imprime el *token* resultante en el archivo *output.txt*. En el caso de un error léxico, se ejecuta el siguiente código: Cuando se encuentra un error léxico, no se retorna al analizador sintáctico. Los patrones que se utilizan están en `lexer.py` (líneas 58-103) son una adaptación de la gramática en la tabla 2.1. Note la tupla *tokens*. Esta contiene todos los *tokens* del lenguaje AVISMO. Sin embargo, no todos están definidos a el nivel léxico. La mayoría de los *tokens* se construyen en la etapa sintáctica, particularmente los que tienen que ver con compuestos químicos y modelos moleculares. Mas aún, note el objeto *tok* en la línea 5 de el listado 2.10. Este contiene los atributos *type*, *value*, *lineno* y *lexpos*. Estos devuelven, respectivamente, el tipo, lexema, en que línea del archivo se encuentra y la posición del primer caracter de el *token* llamado tok. Otro detalle importante de la implementación léxica es el orden de aplicación de los patrones. Los patrones en forma de variables, como en el listado 2.12, tienen que estar escrito antes que los patrones escritos en forma de funciones, como en el listado 2.9 y 2.13

```
t_ENLACE = r"(-|=|:|::)" #define los tokens para
    ↪ diferentes tipos de enlaces quimicos
```

Figura 2.12: Patrón en forma de variable

```

def t_ELEMENTO_QUIMICO(t):  #define regla para el
    ↪ token elemento quimico

    r"(H|Li|Na|K|Rb|Cs|Fr|Be|Mg|Ca|Sr|Ba|Ra|Sc|Y|Ti|Z|
    ↪ r|Hf|Db|V|Nb|Ta|Ji|Cr|Mo|W|Rf|Mn|Tc|Re|Bh|Fe|_
    ↪ Ru|Os|Hn|Co|Rh|Ir|Mt|Ni|Pd|Pt|Cu|Ag|Au|Zn|Cd|_
    ↪ Hg|B|Al|Ga|In|Ti|C|Si|Ge|Sn|Pb|N|P|As|Sb|Bi|O|
    ↪ |S|Se|Te|Po|F|Cr|Br|I|At|He|Ne|Ar|Kr|Xe|Rn)"

    return t

```

Figura 2.13: Patrón en forma de función

Capítulo 3

Analizador Sintáctico

3.1 *grammar.py*

Un código fuente pasa por al menos dos fases:

1. Análisis léxico
2. Análisis sintáctico

Como se ha mencionado antes, en esta primera fase se evalúa el código fuente carácter a carácter. Este se tokeniza, es decir se le otorga una categoría sintáctica, y se devuelve al analizador sintáctico. Ahora, la cuestión es, cuál es el propósito del analizador sintáctico? El analizador sintáctico recibe una lista de *tokens* del analizador léxico y las convierte, a través de reglas de producción, en sentencias gramaticales del lenguaje en cuestión. Reglas de producción tienen la siguiente forma: (**Regla** : *Definición*) donde **Regla** es un no terminal y *Definición* es una

serie de 0 o mas terminales o no terminales. Un terminal se define como un *token* y un no terminal es un una regla gramatical en si.

Se utiliza PLY (*PLY (Python Lex-Yacc) — Ply 4.0 Documentation*, n.d.) para el análisis sintáctico. En particular, su implementación de *yacc* (*Man Yacc (1): An LALR(1) Parser Generator*, n.d.). En el archivo *grammar.py* se puede apreciar que la todas de las reglas del lenguaje AVISMO, con la exención de reglas que fueron utilizadas en analizador léxico, fueron adaptadas. En PLY, una regla gramatical se define como una función en Python cuyo nombre es **p_** seguido del nombre de la regla de producción, por ejemplo:

```
1 def p_s(p) :  
2     '''s : INICIO sentencias FIN'''
```

Figura 3.1: Ejemplo de una regla de producción en PLY

Note que el argumento *p* es una lista que contiene objetos *LexToken*. Los terminales tienen una variable de valor asignada mientras que los no terminales no. Cada objeto *LexToken* tiene una posición léxica (como una variable miembro llamada *lexpos*) y la línea dentro del código fuente (como una variable miembro llamada *lineno*). La manera de definir la regla de producción se puede ver en la figura 3.1, línea 2. Esta sigue el formato previamente establecido pero con un detalle importante. Los no terminales están escritos en letras minúsculas y los terminales en mayúsculas. Esto se hizo con el motivo de clarificar en que categoría, si terminal o no terminal, es clasificada cada ítem en la regla de producción. Más

aún, la documentación de *PLY* sugiere esta convención.

Toda gramática parte desde un axioma y *PLY* sigue este principio. Por defecto, *PLY* asume que la primera regla que se define en el archivo de *grammar.py* es el axioma de la gramática. Sin embargo, es preferible que se defina un axioma explícito. En *PLY*, si se le asigna a la variable *start* el nombre de la regla de producción como una cadena de caracteres, como se hace a continuación, `start = "s"`, *PLY* explícitamente comienza la derivación desde esa regla. Declarar el axioma explícitamente tiene dos ventajas:

- Claridad en el código
- Eliminación de errores por tokens no utilizados

Debido a que no se está implementando la funcionalidad del lenguaje AVISMO, las reglas de producción no tienen código relevante. Sin embargo, todas las reglas de producción en *grammar.py* ejecutan una función llamada *format_expr* que guarda información acerca de la regla gramatical que se utilizó en la derivación del código fuente. Esto se hace con intenciones pedagógicas. El código de *format_expr* se presenta a continuación:

```

1 def format_expr(p):
2     types = [x.type for x in p.slice]
3     rule = f"{types[0]} --> "
4     for val in types[1:]:
5         rule += f"{val} "
6     rules.append([rule])

```

Figura 3.2: Código para guardar información acerca de las derivaciones

3.2 *tester.py*

Para poder correr (grammar.py) en un archivo escrito en AVISMO, es necesario invocar de correr el programa *tester.py* de la siguiente manera:

```
python3 tester.py archivo
```

donde archivo es un programa de AVISMO. Al invocar el comando anterior, se presenta la derivación *LALR* del código fuente. Para un ejemplo, vease el código AVISMO en las figuras en el apéndice C.

Capítulo 4

Conclusiones y Recomendaciones

Inicialmente, el analizador léxico de este proyecto fue escrito con *Flex* (*Flex - a Scanner Generator*, n.d.) en C++. Esto se hizo porque nosotros no habíamos escrito en C++ en mucho tiempo y deseábamos elaborar un proyecto extenso en el para mejorar nuestro entendimiento del lenguaje. Sin embargo, se tuvo que abandonar este camino debido a una combinación de limitaciones de tiempo, documentación pobre, pocos recursos de donde tomar inspiración, etc. Debido a esta situación se tuvo que re-escribir el analizador léxico en *PLY*, la cual tiene documentación mejor, recursos extensos, buen ejemplos, entre otros beneficios. De este cambio se aprendieron varias lecciones. Entre ellas la importancia de utilizar la herramienta mas apropiada para el trabajo. En el desarrollo de *software*, es importante utilizar herramientas que tengan una base amplia de soporte, independientemente de las metas personales de los diseñadores.

En conclusión, este proyecto fue una experiencia fascinante y despertadora.

Como programadores, los compiladores y interpretadores son nuestras herramientas de uso diario, como es el martillo para un carpintero. A veces se nos olvida que los lenguaje de programación están diseñadas para ser escritos y entendidos por humanos porque su estructura parece tan disimilar a las lenguas naturales. Gracias a esta experiencia, dimos un paso hacia atrás y pudimos apreciar lo complejo que es diseñar un analizador léxico y sintáctico. Creemos que jamas perderemos la paciencia con un error de compilación. Sino nos sentiremos agradecidos algún programador tomo el tiempo de crear la herramientas que no tan solo nos provee un sueldo. No creemos que sea una exageración decir que gracias a la labor colaborativa de muchos académicos a traves del tiempo, han cambiado el mundo, una línea de código a la vez.

Appendix A

Código de Analizador Léxico

```
1  import os
2  import sys
3
4  import ply.lex as lex
5  from prettytable import PrettyTable
6
7
8  reserved = {}
9
10 variables = {}
11
12 with open("keywords.txt", "r") as f:
13     for line in f:
```

```

14         val = line.strip()
15         reserved[val] = val.upper()
16
17     try:
18         test_file = sys.argv[1]
19     except IndexError:
20         test_file = "test_prog.txt"
21
22     if not (os.path.exists(test_file)):
23         print(
24             f"The file {test_file} not found, proceeding
25             ↪ with default\
26             test_prog.txt file")
27         test_file = "test_prog.txt"
28
29 tokenTable = PrettyTable()
30
31 # enumera los nombre de todos tokens que puede
32 ↪ reconocer
33 tokens = [
34     "FIN_DE_LINEA",
35     # "LETRA",
36     # "DIGITO",

```

```

35     "OPERACION",
36     "VALENCIA",
37     "ENLACE",
38     # "IDCONT",
39     "ID",
40     "ELEMENTO_QUIMICO",
41     # "MODELO_MOLECULAR",
42     # "COMPUESTO",
43     # "COMPUESTOS",
44     # "ELEMENTO",
45     # "GRUPO_FUNCIONAL",
46     # "GRUPO_FUNCIONAL_INFERIOR",
47     # "GRUPO_FUNCIONAL_SUPERIOR",
48     # "MODELO_GRUPO_FUNCIONAL",
49     # "SENTENCIA",
50     # "SENTENCIAS",
51     "PARENTESIS_IZQ",
52     "PARENTESIS_DER",
53     "TIPO",
54     # "PALABRA_RESERVADA",
55     "COR_IZQ",
56     "COR_DER",
57     "ASIGNACION",

```

```

58 ]
59
60 tokens = tokens + list(reserved.values())
61
62
63 # definiciones de los tokens y reglas de expresiones
    ↳ regulares
64 # t_COR_IZQ y t_COR_DER definen los tokens para
    ↳ corchetes izquierdos y
65 # derechos [ y ]
66 t_COR_IZQ = r"\["
67 t_COR_DER = r"\]"
68 # t_PARENTESIS_IZQ y t_PARENTESIS_DER definen los
    ↳ tokens para parentesis
69
70 # izquierdos y derechos ( y )
71 t_PARENTESIS_IZQ = r"\("
72 t_PARENTESIS_DER = r"\)"
73
74 # define el token para el final de la linea, que
    ↳ puede ser : o ;
75 t_FIN_DE_LINEA = r"(:|;)"

```

```

76 # define los tokens para cualquier numero entero del
    ↳ 1 al 9
77 t_VALENCIA = r"[1-9]"
78 # t_DIGITO = r"[0-9]" # define los tokens para
    ↳ cualquier digito del 0 al 9
79 t_TIPO = r"modelo" # define el token para la palabra
    ↳ "modelo"
80 # define los tokens para diferentes tipos de enlaces
    ↳ quimicos
81 t_ENLACE = r"(-|=|:|::)"
82 t_ignore = " \t" # indica que se deben ignorar los
    ↳ espacios en blanco y
83 # tabulaciones
84
85
86 def t_ASIGNACION(t): # identifica el token "!="
87     r"!="
88     return t
89
90 # identifica el token graficar2d, graficar3d y
    ↳ pesomolecular
91
92

```

```

93 def t_OPERACION(t):
94     r"(graficar2d|graficar3d|pesomolecular)"
95     return t
96
97
98 def t_ELEMENTO_QUIMICO(t): # define regla para el
    ↪ token elemento quimico
99     r"(H|Li|Na|K|Rb|Cs|Fr|Be|Mg|Ca|Sr|Ba|Ra|Sc|Y|Ti|Z|
    ↪ r|Hf|Db|V|Nb|Ta|Ji|Cr|Mo\
100         |W|Rf|Mn|Tc|Re|Bh|Fe|Ru|Os|Hn|Co|Rh|Ir|Mt|Ni|
    ↪ Pd|Pt|Cu|Ag|Au|Zn|Cd|Hg|B\
101         |Al|Ga|In|Ti|Cl|Si|Ge|Sn|Pb|N|P|As|Sb|Bi|O|S|
    ↪ Se|Te|Po|F|C|Br|I|At|He|\
102         Ne|Ar|Kr|Xe|Rn)"
103     return t
104
105 # identifica el token ID pero tambien idetifica el
    ↪ token de palabra reservada
106
107
108 def t_ID(t):
109     r"[A-Za-z]+\d*"
110     isPR = reserved.get(t.value, "ID")

```



```

111     # if isPR != "ID":
112     #     t.type = "PALABRA_RESERVADA"
113     # else:
114     #     t.type = isPR
115     #     variables[t.value] = ""
116     if isPR != "MODELO":
117         t.type = isPR
118     elif isPR == "MODELO":
119         t.type = "TIPO"
120     if isPR == "ID":
121         variables[t.value] = ""
122     return t
123
124
125 def t_newline(t):                # incrementa el numero de
    ↪ linea
126     r'\n+'
127     t.lexer.lineno += len(t.value)
128
129
130 def t_COMMENT(t):                # Ignora comentarios
131     r'\#.*'
132     pass

```

```

133     # No return value. Token discarded
134
135
136     def t_error(t):                # identifica error lexico
137         tokenTable.add_row([tokenNum, "ERROR", t.value[0],
138                             t.lineno, t.lexpos, test_file])
139         t.lexer.skip(1)
140
141
142     lexer = lex.lex()
143
144     tokenNum = 1
145     if __name__ == "__main__":
146         tokenTable.field_names = ["N.", "Token",
147                                   "Lexema", "Linea",
148                                   ↳ "Posicion",
149                                   ↳ "Programa"]
150
151         reservedWords = PrettyTable()
152         reservedWords.field_names = ["Palabra Reservada"]
153         symbolsTable = PrettyTable()
154         symbolsTable.field_names = ["Variables"]
155
156         with open(test_file, "r") as f:

```

```

154     with open("output.txt", "w") as o:
155         for data in f:
156             # data = input("Input data: ")
157             lexer.input(data)
158             for tok in lexer:
159                 tokenTable.add_row(
160                     [tokenNum, tok.type,
161                      ↪ tok.value, tok.lineno,
162                      ↪ tok.lexpos,
163                      test_file])
164                 tokenNum += 1
165                 # o.write(line+"\n")
166             o.write(str(tokenTable))
167             line = "\n\nTABLA DE SIMBOLOS"
168             o.write(line+"\n")
169             for val in variables:
170                 symbolsTable.add_row([val])
171             o.write(str(symbolsTable)+"\n")
172             line = "\n\nPALABRAS RESERVADAS"
173             o.write(line+"\n")
174             for val in reserved:
175                 reservedWords.add_row([val])
176             o.write(str(reservedWords)+"\n")

```

175

176 **with** **open**("output.txt", "r") **as** f:

177 **for** line **in** f:

178 **print**(line, end="")

Appendix B

Código de Analizador Sintáctico

```
1  from lexer import tokens
2  from lexer import variables
3  from ply import yacc
4  import sys
5
6  rules = []
7
8
9  def format_expr(p):
10     types = [x.type for x in p.slice]
11     rule = f"{types[0]} --> "
12     for val in types[1:]:
13         rule += f"{val} "
```

```

14     rules.append([rule])
15     # print(p.lexspan(1))
16
17
18     start = "s"
19
20
21     def p_s(p):
22         '''s : INICIO sentencias FIN'''
23         # p[0] = p[2]
24         format_expr(p)
25
26
27     def p_sentencias(p):
28         '''sentencias : sentencia FIN_DE_LINEA sentencias
29                        | sentencia FIN_DE_LINEA'''
30         # if (len(p) == 4):
31         #     p[0] = p[1] + p[3]
32         # elif (len(p) == 3):
33         #     p[0] = p[1]
34         format_expr(p)
35
36

```

```

37 def p_sentencia(p):
38     '''sentencia : DEFINA ID COMO TIPO
39                     | ID ASIGNACION modelo_molecular
40                     | OPERACION PARENTESIS_IZQ ID
    ↪ PARENTESIS_DER'''
41     # type1 = p[1].type
42     # print(type1)
43     format_expr(p)
44     # if (len(p) == 4):
45     #     print(p[1])
46     #     print(p[3])
47     #     variables[p[1]] = p[3]
48
49
50 def p_modelo_molecular(p):
51     '''modelo_molecular : ELEMENTO_QUIMICO
52                           | ELEMENTO_QUIMICO VALENCIA
53                           | elemento grupo_funcional
54                           | compuesto elemento
55                           | compuesto elemento
    ↪ grupo_funcional
56                           | compuesto compuesto
    ↪ compuestos'''

```

```

57     format_expr(p)
58
59
60     def p_compuesto(p) :
61         '''compuesto : ELEMENTO_QUIMICO
62                     | ELEMENTO_QUIMICO VALENCIA
63                     | elemento grupo_funcional
64                     | elemento grupo_funcional ENLACE
65                     | elemento ENLACE'''
66     format_expr(p)
67
68
69     def p_compuestos(p) :
70         '''compuestos : compuesto compuestos
71                     | compuesto'''
72     format_expr(p)
73
74
75     def p_elemento(p) :
76         '''elemento : ELEMENTO_QUIMICO
77                     | ELEMENTO_QUIMICO VALENCIA'''
78     format_expr(p)
79

```



```

80
81 def p_grupo_funcional(p):
82     '''grupo_funcional : grupo_funcional_inferior
      ↪ grupo_funcional_superior
83         | grupo_funcional_superior
      ↪ grupo_funcional_inferior
84         | PARENTESIS_IZQ
      ↪ modelo_grupo_funcional PARENTESIS_DER
85         | COR_IZQ
      ↪ modelo_grupo_funcional COR_DER'''
86     format_expr(p)
87
88
89 def p_grupo_funcional_inferior(p):
90     '''grupo_funcional_inferior : COR_IZQ
      ↪ modelo_grupo_funcional COR_DER'''
91     format_expr(p)
92
93
94 def p_grupo_funcional_superior(p):
95     '''grupo_funcional_superior : PARENTESIS_IZQ
      ↪ modelo_grupo_funcional PARENTESIS_DER'''
96     format_expr(p)

```

```

97
98
99 def p_modelo_grupo_funcional(p):
100     '''modelo_grupo_funcional : ENLACE
        ↳ modelo_molecular
101                                     | ELEMENTO_QUIMICO
102                                     | ELEMENTO_QUIMICO
        ↳ VALENCIA
103                                     | elemento
        ↳ grupo_funcional
104                                     | compuesto elemento
105                                     | compuesto elemento
        ↳ grupo_funcional
106                                     | compuesto compuesto
        ↳ compuestos'''
107     format_expr(p)
108
109
110 def find_column(input, token):
111     line_start = input.rfind('\n', token.lineno,
        ↳ token.lexpos) + 1
112     return (token.lexpos - line_start) + 1
113

```

```

114
115 try:
116     test_file = sys.argv[1]
117 except IndexError:
118     test_file = "test_prog.txt"
119
120
121 def p_error(p):
122     if p:
123         with open(test_file, "r") as f:
124             line = f.read()
125             err = f"Error sintactico en la linea
126                 ↪ {p.lineno}, columna {find_column(line,
127                 ↪ p)}\n"
128             por {p.type}\n"
129             with open("parser_err_out.txt", "a") as f:
130                 f.write(err)
131                 print(err)
132                 parser.errok()
133     else:
134         print("Error Sintactico en el final del
135             ↪ archivo")
136
137
138

```

```
134
135 parser = yacc.yacc(debug=True)
136 if __name__ == "__main__":
137     while True:
138         try:
139             s = input('calc > ')
140         except EOFError:
141             break
142         if not s:
143             continue
144         parser.parse(s)
```

Appendix C

Casos de Prueba

```
1 inicio
2 defina comp1 como modelo;
3 comp1 := C:H3CH(Cl2)[KP7]=CH3;
4 graficar2d(comp1);
5 fin

● 88 comp4999_compilers_project/code/python_remake/test_prog.txt 1:0 All
~/m/D/c/code/python_remake [master] λ python3 tester.py test_prog.txt
+-----+
| Regla |
+-----+
| sentencia --> DEFINA ID COMO TIPO
| elemento --> ELEMENTO_QUIMICO
| compuesto --> elemento ENLACE
| compuesto --> ELEMENTO_QUIMICO VALENCIA
| compuesto --> ELEMENTO_QUIMICO
| elemento --> ELEMENTO_QUIMICO
| modelo_grupo_funcional --> ELEMENTO_QUIMICO VALENCIA
| grupo_funcional_superior --> PARENTESIS_IZQ modelo_grupo_funcional PARENTESIS_DER
| compuesto --> ELEMENTO_QUIMICO
| elemento --> ELEMENTO_QUIMICO VALENCIA
| modelo_grupo_funcional --> compuesto elemento
| grupo_funcional_inferior --> COR_IZQ modelo_grupo_funcional COR_DER
| grupo_funcional --> grupo_funcional_superior grupo_funcional_inferior
| compuesto --> elemento grupo_funcional ENLACE
| compuesto --> ELEMENTO_QUIMICO
| compuesto --> ELEMENTO_QUIMICO VALENCIA
| compuestos --> compuesto
| compuestos --> compuesto compuestos
| compuestos --> compuesto compuestos
| compuestos --> compuesto compuestos
| modelo_molecular --> compuesto compuesto compuestos
| sentencia --> ID ASIGNACION modelo_molecular
| sentencia --> OPERACION PARENTESIS_IZQ ID PARENTESIS_DER
| sentencias --> sentencia FIN_DE_LINEA
| sentencias --> sentencia FIN_DE_LINEA sentencias
| sentencias --> sentencia FIN_DE_LINEA sentencias
| s --> INICIO sentencias FIN
+-----+
```

Figura C.1: Ejemplo de Programa Correcto

```
1 inicio
2 defina a1 como modelo;
3 a1 := CH3CH(CH3)CH3;
4 fin
```

57 comp4999_compilers_project/code/python_remake/test_prog2.txt 5:0 All

doom:eshell-popup:#1 Fri Apr 21 13:01:09 2023

~/m/D/c/code/python_remake [master] λ python3 tester.py test_prog2.txt

```
+-----+
| Regla                                     |
+-----+
| sentencia --> DEFINA ID COMO TIPO        |
| compuesto --> ELEMENTO_QUIMICO           |
| compuesto --> ELEMENTO_QUIMICO VALENCIA  |
| compuesto --> ELEMENTO_QUIMICO           |
| elemento --> ELEMENTO_QUIMICO            |
| compuesto --> ELEMENTO_QUIMICO           |
| elemento --> ELEMENTO_QUIMICO VALENCIA   |
| modelo_grupo_funcional --> compuesto elemento |
| grupo_funcional --> PARENTESIS_IZQ modelo_grupo_funcional PARENTESIS_DER |
| compuesto --> elemento grupo_funcional    |
| compuesto --> ELEMENTO_QUIMICO           |
| compuesto --> ELEMENTO_QUIMICO VALENCIA   |
| compuestos --> compuesto                 |
| compuestos --> compuesto compuestos      |
| compuestos --> compuesto compuestos      |
| compuestos --> compuesto compuestos      |
| modelo_molecular --> compuesto compuesto compuestos |
| sentencia --> ID ASIGNACION modelo_molecular |
| sentencias --> sentencia FIN_DE_LINEA    |
| sentencias --> sentencia FIN_DE_LINEA sentencias |
| s --> INICIO sentencias FIN              |
+-----+
```

Figura C.2: Ejemplo de Programa Correcto

```
1 inicio
2 defina id como modelo;
3 pesomolecular(id);
4 graficar2d(id);
5 fin

● 70 comp4999_compilers_project/code/python_remake/test_prog4.txt 4:12 All
*doom:eshell-popup:#1* Fri Apr 21 13:19:22 2023

~/m/D/c/code/python_remake [master] λ python3 tester.py test_prog4.txt
+-----+
| Regla |
+-----+
| sentencia --> DEFINA ID COMO TIPO |
| sentencia --> OPERACION PARENTESIS_IZQ ID PARENTESIS_DER |
| sentencia --> OPERACION PARENTESIS_IZQ ID PARENTESIS_DER |
| sentencias --> sentencia FIN_DE_LINEA |
| sentencias --> sentencia FIN_DE_LINEA sentencias |
| sentencias --> sentencia FIN_DE_LINEA sentencias |
| s --> INICIO sentencias FIN |
+-----+
```

Figura C.3: Ejemplo de Programa Correcto


```

1 inicio
2 defina comp1 como modelo;
3 comp1 = := C:H3CH(Cl2)[KP7]=CH3;=
4 graficar2d(comp1);
5 fin;

```

92 comp4999_compilers_project/code/python_remake/test_prog1.txt 6:0 All

doom:eshell-popup:#1 Fri Apr 21 12:57:39 2023

~/m/D/c/code/python_remake [master] λ python3 tester.py test_prog1.txt

Error sintactico en la linea 3, columna 7 por ENLACE

Error sintactico en la linea 3, columna 33 por ENLACE

Error sintactico en la linea 5, columna 4 por FIN_DE_LINEA

```

+-----+
| Regla |
+-----+
| sentencia --> DEFINA ID COMO TIPO
| elemento --> ELEMENTO_QUIMICO
| compuesto --> elemento ENLACE
| compuesto --> ELEMENTO_QUIMICO VALENCIA
| compuesto --> ELEMENTO_QUIMICO
| elemento --> ELEMENTO_QUIMICO
| modelo_grupo_funcional --> ELEMENTO_QUIMICO VALENCIA
| grupo_funcional_superior --> PARENTESIS_IZQ modelo_grupo_funcional PARENTESIS_DER
| compuesto --> ELEMENTO_QUIMICO
| elemento --> ELEMENTO_QUIMICO VALENCIA
| modelo_grupo_funcional --> compuesto elemento
| grupo_funcional_inferior --> COR_IZQ modelo_grupo_funcional COR_DER
| grupo_funcional --> grupo_funcional_superior grupo_funcional_inferior
| compuesto --> elemento grupo_funcional ENLACE
| compuesto --> ELEMENTO_QUIMICO
| compuesto --> ELEMENTO_QUIMICO VALENCIA
| compuestos --> compuesto
| compuestos --> compuesto compuestos
| compuestos --> compuesto compuestos
| compuestos --> compuesto compuestos
| modelo_molecular --> compuesto compuesto compuestos
| sentencia --> ID ASIGNACION modelo_molecular
| sentencia --> OPERACION PARENTESIS_IZQ ID PARENTESIS_DER
| sentencias --> sentencia FIN_DE_LINEA
| sentencias --> sentencia FIN_DE_LINEA sentencias
| sentencias --> sentencia FIN_DE_LINEA sentencias
| s --> INICIO sentencias FIN
+-----+

```

Figura C.4: Ejemplo de Programa Incorrecto

```
1 inicio
2 defina comp1 como modelo;
3 comp1 = CH3CH(CH3)CH3;
4 fin

● 61 comp4999_compilers_project/code/python_remake/test_prog3.txt 5:0 All
~/m/D/c/code/python_remake [master] λ python3 tester.py test_prog3.txt
Error sintactico en la linea 3, columna 7      por ENLACE
Error sintactico en la linea 3, columna 9      por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 10     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 11     por VALENCIA
Error sintactico en la linea 3, columna 12     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 13     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 14     por PARENTESIS_IZQ
Error sintactico en la linea 3, columna 15     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 16     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 17     por VALENCIA
Error sintactico en la linea 3, columna 18     por PARENTESIS_DER
Error sintactico en la linea 3, columna 19     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 20     por ELEMENTO_QUIMICO
Error sintactico en la linea 3, columna 21     por VALENCIA
Error sintactico en la linea 3, columna 22     por FIN_DE_LINEA
Error sintactico en la linea 4, columna 1     por FIN

Error Sintactico en el final del archivo
+-----+
| Regla |
+-----+
| sentencia --> DEFINA ID COMO TIPO |
+-----+
```

Figura C.5: Ejemplo de Programa Incorrecto

```
1 inicio KCH;
2 defina id como modelo;
3 pesomolecular(id);
4 graficar2d(id);
5 fin

● 74 comp4999_compilers_project/code/python_remake/test_prog5.txt 6:0 All
~/m/D/c/code/python_remake [master] λ python3 tester.py test_prog5.txt
Error sintactico en la linea 1, columna 8      por ELEMENTO_QUIMICO

Error sintactico en la linea 1, columna 9      por ELEMENTO_QUIMICO

Error sintactico en la linea 1, columna 10     por ELEMENTO_QUIMICO

Error sintactico en la linea 1, columna 11     por FIN_DE_LINEA

+-----+
| Regla |
+-----+
| sentencia --> DEFINA ID COMO TIPO
| sentencia --> OPERACION PARENTESIS_IZQ ID PARENTESIS_DER
| sentencia --> OPERACION PARENTESIS_IZQ ID PARENTESIS_DER
| sentencias --> sentencia FIN_DE_LINEA
| sentencias --> sentencia FIN_DE_LINEA sentencias
| sentencias --> sentencia FIN_DE_LINEA sentencias
| s --> INICIO sentencias FIN
+-----+
```

Figura C.6: Ejemplo de Programa Incorrecto

Lista de Figuras

2.1	Automata del patrón para el token <FIN_DE_LINEA>	13
2.2	Automata del patrón para el token <PALABRAS_RESERVADA>	13
2.3	Automata del patrón para el token <ID>	13
2.4	Automata del patrón para el token <IDCONT>	13
2.5	Automata del patrón para el token <ASIGNACION>	14
2.6	Automata del patrón para el token <LETRA>	14
2.7	Automata del patrón para el token <DIGITO>	14
2.8	Tabla de símbolos implementada como un diccionario	14
2.9	Patrón que ceeza identificadores de palabras reservadas	16
2.10	Código para imprimir los <i>tokens</i> encontrados	17
2.11	Gestión de errores	18
2.12	Patrón en forma de variable	18
2.13	Patrón en forma de función	19
3.1	Ejemplo de una regla de producción en PLY	21
3.2	Código para guardar información acerca de las derivaciones	23

C.1	Ejemplo de Programa Correcto	45
C.2	Ejemplo de Programa Correcto	46
C.3	Ejemplo de Programa Correcto	47
C.4	Ejemplo de Programa Incorrecto	48
C.5	Ejemplo de Programa Incorrecto	49
C.6	Ejemplo de Programa Incorrecto	50

Lista de Tablas

2.1	Tabla de Componentes Léxicos de AVISMO	12
-----	--	----

Referencias Bibliográficas

Flex - a scanner generator. (n.d.). https://ftp.gnu.org/old-gnu/Manuals/flex-2.5.4/html_mono/flex.html.

Man yacc (1): An LALR(1) parser generator. (n.d.). <https://manpages.org/yacc>.

Narciso Farias, F., Rios, A., Hidrobo, F., & Vicuña, O. (2012, May). Una gramática libre de contexto para el lenguaje del ambiente de visualización molecular - AVISMO..

PL-Project-LGM-YVV-AMN/MAPL. (n.d.). <https://github.com/PL-Project-LGM-YVV-AMN/MAPL>.

PLY (Python Lex-Yacc) — ply 4.0 documentation. (n.d.). <https://ply.readthedocs.io/en/latest/index.html>.