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

Aramis E. Matos

Lenier Gerena

Angel Berrios Pellot

Segundo Semestre, 2022-2023

Tabla de Contenido

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 desarollo 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

- <SENTENCIAS> ::= <FIN_DE_LINEA> <SENTENCIAS> | <SENTENCIA> <FIN_DE_LINEA>
- <FIN_DE_LINEA> ::= ":" | ";"
- <SENTENCIA> ::= "defina" <ID> "como" <TIPO> | <ID> "=" <MODELO_MOLECULAR> | <OPERACION> "(" <ID> ")"
- <ID> ::= "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>

- <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_GRUYPO_FUNCIONAL> ")
- <MODELO_GRUYPO_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> <VALENCIA> | <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.

Token	Patrón	Lexema	Atributos
<FIN_DE_LINEA>	; | :	:	Simbolo 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" | "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")	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

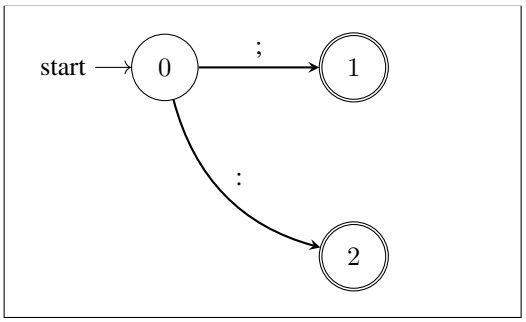

Figura 2.1: Automata del patrón para el token <FIN_DE_LINEA>

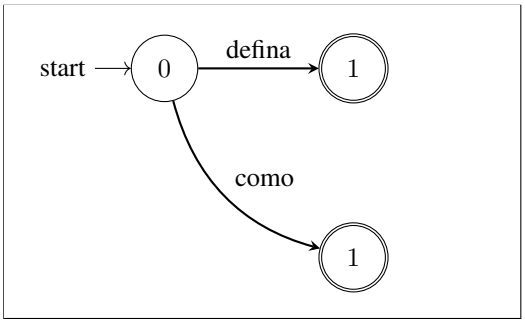

Figura 2.2: Automata del patrón para el token <PALABRAS_RESERVADA>

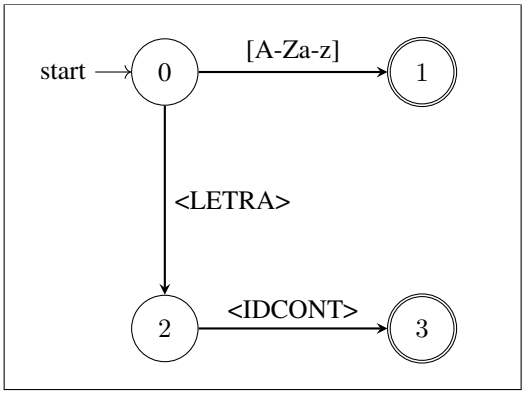

Figura 2.3: Automata del patrón para el token <ID>

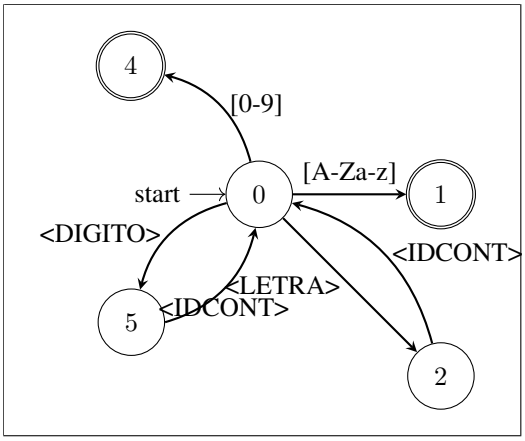

Figura 2.4: Automata del patrón para el token <IDCONT>

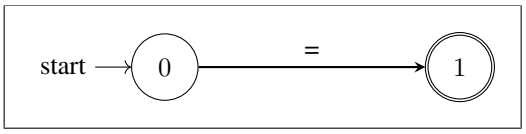

Figura 2.5: Automata del patrón para el token <ASIGNACION>

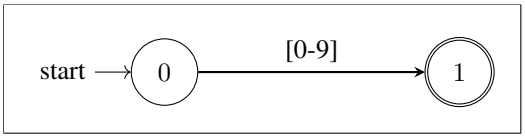

Figura 2.6: Automata del patrón para el token <LETRA>

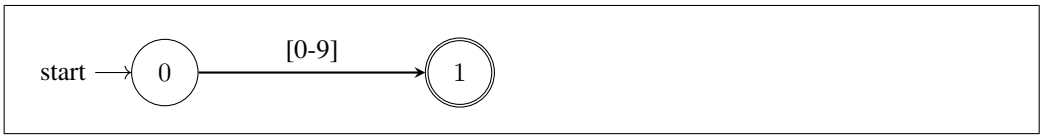

Figura 2.7: Automata del patrón para el token <DIGITO>

2.2.2 Tabla de Símbolos

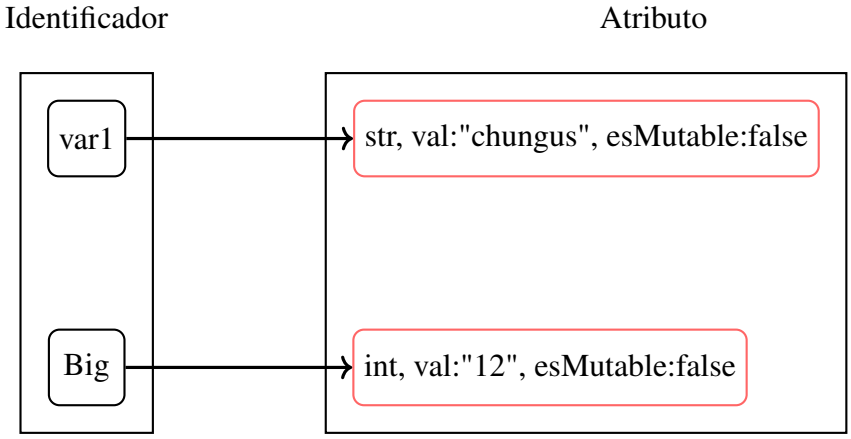

Figura 2.8: Tabla de símbolos implementada como un diccionario

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 ambiguedad semántica debido a que la gramatica 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:

```
with open(test_file, "r") as f:
    with open("output.txt","w") as o:
        for data in f:
            lexer.input(data)
            for tok in lexer:
                tokenTable.add_row([tokenNum,tok.type
                 ↪  ,tok.value,tok.lineno,tok.lexpos,
                 ↪  test_file])
                tokenNum += 1
        o.write(str(tokenTable))
        line = "\n\nTABLA DE SIMBOLOS"
        o.write(line+"\n")
        for val in variables:
            symbofigable.add_row([val])
        o.write(str(symbofigable)+"\n")
        line = "\n\nPALABRAS RESERVADAS"
        o.write(line+"\n")
        for val in reserved:
            reservedWords.add_row([val])
        o.write(str(reservedWords)+"\n")
```

Figura 2.10: Código para imprimir los *tokens* encontrados

```
def t_error(t):                 # identifica error lexico
    tokenTable.add_row([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 quimicos 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

Implementación del 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 traves 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, linea 2. Esta sigue el formato previamente establecido pero con un detalle importante. Los no terminales están escritos en letras minúsculas y los ter-

minales en mayúsculas. Esto se hizo con el motivo de clarificar en que categoría, si terminal o no terminal, es clasificada cada item 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 defacto, *PLY* asume que la primera regla que se define en el archivo de *grammar.py* es el axioma del 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:

```
inicio
defina a1 como modelo;
a1 := CH3CH(CH3)CH3;
fin
```

Figura 3.3: Ejemplo de un programa AVISMO

```
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. Por ejemplo, el código AVISMO en la figura 3.3 se deriva de la manera que se presenta en la figura

```
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
```

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. Aveces 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.

Lista de Figuras

Lista de Tablas

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.