DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico:
Compositor de fórmulas matemáticas

Teoría de Lenguajes

Integrante	LU	Correo electrónico
Santos, Martiín	413/11	martin.n.santos@gmail.com
Szyrej, Alexander	642/11	alexander.szyrej@gmail.com
Nuñez Morales, Carlos Daniel	732/08	cdani.nm@gmail.com

Resumen:

Este trabajo consiste en el armado de un archivo SVG (fórmula) en base a un input, una cadena perteneciente al lenguaje de una gramática ambigua definida en el enunciado.

Keywords:

Gramática, Parsers, LALR, AST, SVG, LATEX

Índice

1. Introduccion

Este trabajo consta de dos etapas: Tokenización y parseo del input por un lado (donde también se incluye la desambiguación de la gramática presentada), y por otro lado la generación del archivo de output esperado. Para la primera parte se utilizó el conjunto de herramientas de generación de código python ply. El mismo cuenta con dos módulos, ambos utilizados: Ley, para el proceso de tokenización y Yacc, para parsear la gramática. Para la segunda parte, se cuenta con una estructura árbol, instanciada en la etapa de parsing del input, construida para simplificar el armado del archivo SVG.

2. Parte I: Tokenizacion y Parseo

2.1. Desambiguando la gramática

Utilizamos los módulos de Lex y Yacc para esta primera parte de tokenización y parsing. Puesto que el parser usado por el módulo Yacc utiliza la técnica LALR, nos enfocamos en un primer momento en desambiguar la gramática para que al momento de construir la tabla de *Action* LALR no haya conflictos.

Gramática inicial G:
E → E E
| E ^ E
| E _ E
| E ^ E _ E
| E _ E ^ E
| E / E
| (E)
| { E }
| c

Primeramente consideramos el orden de presedencia dispuesto en el enunciado. Luego la gramática G1 se rige sobre la siguiente tabla:

Caracter	Presedencia	Asociatividad
/	1	Izq
.	2	Izq
^	3	-
_	3	-
()	4	-
{}	4	-

Luego G_1 quedó determinada por el siguiente conjunto de producciones:
E → E / T | T
T → T F | F

F → **I** ^ **G** | **G** | **I** _ **H** | **H**
H → **I** ^ **I** | **I**
G → **I** _ **I** | **I**
I → (E) | {E} | c

Notar que las producciones subrayadas son recursivas a izquierda, lo cual genera conflictos. Las producciones en negrita generaron otros problemas. En estas se debe factorizar a izquiera para no tener conflictos.

Finalmente la gramática quedó de la pinta G_2:
E → T A
A → / T A | λ
T → F B
B → F B | λ
F → I G
G → ^ I H | _ I L | λ
H → _ I | λ
L → ^ I | λ
I → (E) | {E} | c

donde la recursion a izquierda fue eliminada introduciendo las producciones de A y B, y los conflictos de F G y H fueron resueltos subiendo los casos de G y H hasta F y tomando factor común a izquierda.

Con esta nueva gramática G_2 no ambigua, cuyo lenguaje es el mismo que para L(G), pudimos avanzar sobre el módulo de herramientas Lex & Yacc para proceder al análisis léxico y sintáctico.

2.2. Parsing on Python

Utilizamos *ply* para generar código python. En un primer momento definimos los tokens esperados. Los mismos son los que incluye la gramática, es decir tokens = ['(',')','{','}','^','_','/',c]

donde $'c'$ es un caracter cualquiera por fuera de los anteriores, representado con una expresion regular. (NOTA: asumimos c como caracter alfanumérico. Caracteres ASCII podrían fallar)

Acto seguido, definimos las reglas. Las reglas se mapean una a una con la gramática G_2 definida anteriormente. Para facilitar el armado del archivo SVG, construimos una estructura del tipo árbol denominada Abstract Syntax Tree. Luego, por cada regla definida en el parseo se genera un nuevo nodo en el árbol, de forma que quede definida la estructura del input de forma sencilla para un posterior procesamiento del mismo en la construcción del output.

3. Parte II: Construcción del output SVG

3.1. Aclaraciones especiales

El archivo de salida es un XML especial, que debe interpretarse como formato de imagen vectorial SVG. Los archivos deben visualizarse mediante el navegador **Google Chrome**. Otros medios pueden fallar al querer visualizar correctamente los archivos.

3.2. Armado de un Abstract Syntax Tree

El AST consta de siete nodos distintos:

- Nodo División (*divide*), con dos hijos numerador (A) y denominador (B), más un tercero que modela la barra divisoria ($\frac{A}{B}$)
- Nodo Concatenación (*concat*), con dos hijos A y B modelando la concatenación de A con B (AB)
- Nodo Superíndice (p), con dos hijos A y B modelando a B como superíndice de A (A^B)
- Nodo Subíndice (u), con dos hijos A y B modelando a B como subíndice de A (A_B)
- Nodo Super-Subíndice (pu), con tres hijos A B y C modelando a B como superíndice de A y C como subíndice de A (A^B_C)
- Nodo Paréntesis (*parens*), con tres hijos '(', E, ')'
- Nodo Llaves (*brackets*), con tres hijos '{', E, '}'

Por simplicidad se omitió el nodo Sub-Superíndice aún cuando la gramática lo permite, ya que en caso de llegar a un caso de ese estilo al parsear la entrada, simplemente se invierten los parametros al crear el nodo.

Cada Nodo del árbol tiene los atributos que se setean en cada text tag del xml SVG: x, y, z; con z igual al font-size. Así mismo, nodos particulares como la barra divisoria poseen atributos especiales: x1,x2,y1,y2 por ejemplo. Además, todo nodo conoce a su nodo padre y sus nodos hermanos, puede ver sus atributos e incluso copiarles su información.

Para verificar la perfecta construcción de esta estructura se generó una bateria de tests especificos para comprobar que para ciertos casos el árbol generado se corresponda con el esperado.

3.3. Procesamiento del AST

Para recorrer el AST generado en tiempo de parsing decidimos utilizar preorder y diferenciar en cada paso los casos en los cuales nos paramos sobre un nodo terminal, es decir una hoja, o si es un nodo no-terminal. Por lo general, el caso de los no-terminales sólo baja los atributos a las hojas, para que el procesamiento fino se lleve a cabo en ellas. A excepción del caso de la división, donde el propio nodo padre setea los valores y1 e y2 de la barra, y chequea que se ajuste bien considerando casos en los que el numerador posee un subíndice o el denominador un superíndice Los casos más complejos se llevan a cabo en las hojas. Primeramente se diferencia de acuerdo a que tipo tiene el padre del nodo terminal.

- Terminal de una concatenación: Se copia al siguiente en el preorder el y con más el font-size del padre, y al x se le suma un 60 % del font-size.
- Terminal de un superíndice: Se copia al siguiente en el preorder el x con un aumento igual al caso anterior, el y se reduce en una constante para situarlo por encima del caracter anterior y al fontsize se lo setea como un 70 % del anterior.
- Terminal de un subíndice: Similar al caso anterior, pero con al y se lo aumenta para situar al proximo elemento por debajo del anterior.
- Terminal de un super-subíndice: Aquí se combinan los dos casos anteriores, tratando al primero como el caso 2 y al segundo como el caso 3.
- Terminal de una división: El caso, sin dudas, más complejo. Tanto numerador como denominador, de ser terminales funcionan como si fueran hijos de una concatenación. Quien se encarga de armar la división es el último nodo terminal del *divide*, la barra. Esta en primer lugar baja el denominador y lo situa justo debajo del numerador, arrancando ambos en el mismo x. Calcula las longitudes de ambos para así luego centrar al de menos longitud con respecto al otro, y luego setea sus atributos de barra especiales x1 y x2, siendo estos los extremos del opera dor con mayor longitud.
- Terminal de un nodo paréntesis: Tanto el paréntesis de apertura como el nodo central de ser terminal escriben sobre el siguiente en el preorder los atributos actuales, moviendo el x con el aumento. Es el paréntesis de cierre el que, teniendo ya parseada la estructura central, setea los valores correspondientes al transform para el svg. Para ello, busca en la estructura contenida en las llaves el mínimo y el máximo y, de forma tal de poder calcular que tanto debe estirarse.

- Terminal de un nodo llaves: Similar al caso anterior, es la llave de cierra la que guarda la lógica más importante, la de restaurar la configuración previa. Para ello chequea si el hermano del nodo padre era un sub o un superíndice, de modo tal de poder tomar el font-size y el y que corresponden para pasarle esos valores al próximo nodo. De lo contrario, sólo transmite los datos tal cual los tiene.

Luego de recorrer todo el AST, y teniendo en cada nodo los atributos con los que se completa el text tag del xml SVG, sólo resta en sí armarlo. Para ello usamos otro módulo SVG Builder, comentado a continuación.

3.4. SVG Builder

El módulo SVGBuilder es una simple clase que nos abstrae de la creación del archivo SVG. Utiliza el módulo xml.etree.ElementTree para manejar XMLs. Recorre en preorder el árbol AST y por cada nodo, sabiendo de que tipo es, lo escribe en el XML usando sus atributos, teniendo especial cuidado cuando el nodo es un paréntesis o una barra divisoria.

4. Modo de uso

Para invocar el programa debe irse al directorio */tleng-tp1/src/formula*. Una vez allí debe ejecutarse el archivo *form.py* escribiendo *python form.py*. Se generará un SVG con el nombre *form.svg* en el mismo directorio.

Para correr los tests, ir al directorio */tleng-tp1/tests* y escribir *python test.py -v*. El obetivo de estos tests fue verificar que los AST (Abstract Syntax Trees) fueran generados correctamente a partir de una expresión.

5. Resultados

A continuación, algunos resultados interesantes.

5.1. Caso I

Input: (AVIONES/BARCOS)

$$\left(\frac{\text{AVIONES}}{\text{BARCOS}}\right)$$

La dificultad de este caso recae la división de dos nodos no terminales (concatenaciones), donde la barra debe ajustarse al tamaño del mayor de ellos y se debe centrar al menor. Así mismo, se deben ajustar los paréntensis para abarcar toda la división.

5.2. Caso II

Input: MAR/(AGUA+SAL)

$$\frac{\text{MAR}}{\text{(AGUA+SAL)}}$$

En esta oportunidad, la división de, nuevamente, dos nodos no terminales (concatenaciones), con la diferencia de que los paréntensis solo abarcan al denominador de la misma. Además, en este caso el numerador es aquel de longitud menor y por ende el que debe ser centrado.

5.3. Caso III

Input: VIENTOˆ{FUERTE}/VELERO_{EN+PROBLEMAS}

$$\frac{\text{VIENTO}^{\text{FUERTE}}}{\text{VELERO}_{\text{EN+PROBLEMAS}}}$$

En este caso se incluyen subíndices y superíndices, dentro de una división, que es el nodo problemático por excelencia.

5.4. Caso IV

Input: SUPERˆ{HEROE}_{VILLANO}

$$\text{SUPER}_{\text{VILLANO}}^{\text{HEROE}}$$

Por último, la combinación de sub y superíndices, ambos como concatenaciones encerradas por llaves para agruparlas.

6. Apéndices: Exposición de código

6.1. Apéndice I: código principal - from.py

```
#!/usr/bin/env python

import sys
sys.path.insert(0,"../..")
sys.path.insert(0,"../svg")
sys.path.insert(0,"../tree")
sys.path.insert(0,"../ast")


if sys.version_info[0] >= 3:
    raw_input = input

import ply.lex as lex
import ply.yacc as yacc
import os
from svg import SVGBuilder
from tree import Tree
from tree import Node
from ast import ASTProcessor

class Parser:
    """
    Base class for a lexer/parser that has the rules
    defined as methods
    """
    tokens = ()
    precedence = ()

    def __init__(self, **kw):
        self.debug = kw.get('debug', 0)
        self.names = { }
        try:
            modname = os.path.split(os.path.splitext(
                __file__)[0])[1] + "_" + self.__class__
                .__name__
        except:
            modname = "parser"+"_"+self.__class__.
                __name__
        self.debugfile = modname + ".dbg"
        self.tabmodule = modname + "_" + "parsetab"
```

```
        #print self.debugfile, self.tabmodule

        # Build the lexer and parser
        lex.lex(module=self, debug=0)
        yacc.yacc(module=self,
                  debug=0,
                  debugfile=self.debugfile,
                  tabmodule=self.tabmodule)

    def parse(self, exp):
        ast = Tree(yacc.parse(exp))
        return ast

    def run(self, exp):
        ast = Tree(yacc.parse(exp))
        ast_processor = ASTProcessor()
        ast_processor.process(ast)
        svgB = SVGBuilder()
        svg = svgB.build(ast)
        svg.save('form.svg')
        print "'form.svg' generado exitosamente"

class Form(Parser):

    tokens = (
        'UNDERSCORE', 'POW', 'DIVIDE', 'CHAR',
        'LBRACK', 'RBRACK', 'LPAREN', 'RPAREN',
        )

    # Tokens

    t_UNDERSCORE = r'_'
    t_POW = r'\^'
    t_DIVIDE = r'/'
    t_LBRACK = r'\{'
    t_RBRACK = r'\}'
    t_LPAREN = r'\('
    t_RPAREN = r'\)'
    #t_CHAR = r'[a-zA-Z(0-9)*+-]'
    t_CHAR = r'[a-zA-Z+-]'
    t_ignore = " \t"

    def t_error(self, t):
        print("Illegal character '%s'" % t.value[0])
```

```
        t.lexer.skip(1)

    def p_expr_E(self,p):
        """
        expr_E : expr_T expr_A
        """
        if p[2] != None:
            p[0] = Node('divide', [p[1], p[2], Node('
                barra')])
        else: p[0] = p[1]

    def p_expr_A(self,p):
        """
        expr_A : DIVIDE expr_T expr_A
               | empty
        """
        if p[1] == None: p[0] = None
        elif p[3] == None: p[0] = p[2]
        else:
            p[0] = Node('divide', [p[2], p[3], Node('
                barra')])

    def p_expr_T(self,p):
        """
        expr_T : expr_F expr_B
        """
        if p[2] != None:
            p[0] = Node('concat', [p[1], p[2]])
        else: p[0] = p[1]

    def p_expr_B(self,p):
        """
        expr_B : expr_F expr_B
               | empty
        """
        if p[1] == None: p[0] = None
        elif p[2] == None: p[0] = p[1]
        else:
            p[0] = Node('concat', [p[1], p[2]])

    def p_expr_F(self,p):
        """
        expr_F : expr_I expr_G
        """
```

```
        if p[2] != None:
            if p[2][0] == 'pu':
                p[0] = Node('pu', [p[1], p[2][1], p
                    [2][2]])
            elif p[2][0] == 'p':
                p[0] = Node('p', [p[1], p[2][1]])
            else:
                p[0] = Node('u', [p[1], p[2][1]])
        else:
            p[0] = p[1]

    def p_expr_G(self,p):
        """
        expr_G : POW expr_I expr_H
               | UNDERSCORE expr_I expr_L
               | empty
        """
        if p[1] == None:
            p[0] = None
        elif p[1] == '^':
            if p[3] == None:
                p[0] = ('p', p[2])
            else:
                p[0] = ('pu', p[2], p[3])
        elif p[1] == '_':
            if p[3] == None:
                p[0] = ('u', p[2])
            else:
                p[0] = ('pu', p[3], p[2])

    def p_expr_H(self,p):
        """
        expr_H : UNDERSCORE expr_I
               | empty
        """
        if p[1] != None: p[0] = p[2]
        else: p[0] = None

    def p_expr_L(self,p):
        """
        expr_L : POW expr_I
               | empty
        """
        if p[1] != None: p[0] = p[2]
```

```
        else: p[0] = None

    def p_expr_I(self,p):
        """
        expr_I : LPAREN expr_E RPAREN
               | LBRACK expr_E RBRACK
               | CHAR
        """
        if p[1] == '(':
            p[0] = Node('parens', [Node('('), p[2],
                Node(')')])
        elif p[1] == '{':
            p[0] = Node('brackets', [Node('{'), p[2],
                Node('}')])
        else: p[0] = Node(p[1])

    def p_empty(self,p):
        'empty :'
        pass

    def p_error(self,p):
        raise Exception("Syntax error at '%s'" %p.
            value)

if __name__ == '__main__':
    form = Form()
    form.run(sys.argv[1])
```

6.2. Apéndice II: Abstract Syntax Tree - ast.py

```
#!/usr/bin/env python

import sys
sys.path.insert(0,"../tree")

from tree import Tree
from tree import Node

class ASTProcessor:

    """
    Clase encargada de procesar el AST generado por el
       Parser, cargandolo de informacion
    que luego sera utilizada por el SVGBuilder.
```

```
    """

    def process(self, ast):

        # variables nodo root
        ast.root.attrs['x'] = 0
        ast.root.attrs['y'] = 0
        ast.root.attrs['z'] = 1
        ns = ast.preorder_traversal()
        index = 0
        for node in ns:

            ## NO TERMINALES
            if node.type == 'concat' \
            or node.type == 'p' \
            or node.type == 'u' \
            or node.type == 'pu':
                node.first_child().copy_node_attrs(
                    node)

            ## caso DIVIDE: copiamos las variables del
                nodo al primer operador (numerador)
            ## modificando la variable 'y' de manera
               que quede por encima de la barra
            ## de division.
            elif node.type == 'divide':
                node.first_child().copy_node_attrs(
                    node)
                node.first_child().attrs['y'] = node.
                    attrs['y'] - 0.19
                numerador = ns.index(node.children[1])
                    - 1
                if ns[numerador].parent.type == 'u' or
                    ns[numerador].parent.type =='pu':
                    node.last_child().attrs['y1'] =
                        node.attrs['y'] - 0.28*node.
                        attrs['y'] + 0.1
                else:
                    node.last_child().attrs['y1'] =
                        node.attrs['y'] - 0.28*node.
                        attrs['y']
                node.last_child().attrs['y2'] = node.
                    last_child().attrs['y1']
            elif node.type == 'parens':
```

```
        node.first_child().copy_node_attrs(
            node)
    elif node.type == 'brackets':
        node.first_child().copy_node_attrs(
            node)

    ## NODOS TERMINALES ##
    else:
        ## caso CONCAT: copiamos la
            informacion del nodo actual al
            siguiente
        ## nodo en el PREORDER, modificando la
             variable 'x', de manera que
        ## escriba a continuacion del nodo
            actual.
        if node.parent.type == 'concat':
            if (index+1<len(ns)):
                ns[index+1].attrs['x'] = node.
                    attrs['x']+0.6*node.attrs['
                    z']
                ns[index+1].attrs['y'] = node.
                    attrs['y']
                ns[index+1].attrs['z'] = node.
                    attrs['z']

        ## caso P (superindice)
        elif node.parent.type == 'p':
            ## caso Nodo izquierdo de P:
                seteamos las variables del nodo
                 derecho
            ## (superindice) de manera que
                quede mas arriba y con un size
                mas
            ## pequenio. Tambien nos
                aseguramos de avanzar la
                variable 'x'.
            if (node.left_sibling == None):
                node.right_sibling.attrs['x']
                    = node.attrs['x']+0.6*node.
                    attrs['z']
                node.right_sibling.attrs['y']
                    = node.attrs['y']-0.45
                node.right_sibling.attrs['z']
                    = node.attrs['z']*0.7
```

```
            ## caso Nodo derecho de P:
               reestablecemos los valores de '
               y' y 'z'
            ## para que el siguiente nodo del
               PREORDER escriba como inidicaba
            ## la configuracion antes de poner
                el superindice.
            else:
                if (index+1<len(ns)):
                    ns[index+1].attrs['x'] =
                       node.attrs['x']+0.6*
                       node.attrs['z']
                    ns[index+1].attrs['y'] =
                       node.parent.attrs['y']
                    ns[index+1].attrs['z'] =
                       node.parent.attrs['z']


        ## caso U (subindice): parecido al
           caso P con la diferencia de que
           debe escribirse
        ## el subindice por debajo. (variable
           'y')
        elif node.parent.type == 'u':
            ## caso nodo izquierdo de U.
            if (node.left_sibling == None):
                node.right_sibling.attrs['x']
                   = node.attrs['x']+0.6*node.
                   attrs['z']
                node.right_sibling.attrs['y']
                   = node.attrs['y']+0.25
                node.right_sibling.attrs['z']
                   = 0.7*node.attrs['z']
            ## caso nodo derecho de U.
            else:
                if (index+1<len(ns)):
                    ns[index+1].attrs['x'] =
                       node.attrs['x']+0.6*
                       node.attrs['z']
                    ns[index+1].attrs['y'] =
                       node.parent.attrs['y']
                    ns[index+1].attrs['z'] =
                       node.parent.attrs['z']
```

```
## caso PU (superindice y subindice)
elif node.parent.type == 'pu':
    ## caso nodo izquierdo de PU.
       Seteamos las varibles del nodo
       superindice
    ## y subindice
    if (node.left_sibling == None):
        node.right_sibling.attrs['x']
           = node.attrs['x']+0.6*node.
           attrs['z']
        node.right_sibling.attrs['y']
           = node.attrs['y']-0.45
        node.right_sibling.attrs['z']
           = 0.7*node.attrs['z']
        node.last_sibling().attrs['x']
            = node.attrs['x']+0.6*node
           .attrs['z']
        node.last_sibling().attrs['y']
            = node.attrs['y']+0.25
        node.last_sibling().attrs['z']
            = 0.7*node.attrs['z']
    ## caso nodo derecho de PU:
       restablecemos los valores de 'y
       ' y 'z'
    ## y avanzamos la variable 'x'
    elif (node.right_sibling == None):
        if (index+1<len(ns)):
            ns[index+1].attrs['x'] =
               node.attrs['x']+0.6*
               node.attrs['z']
            ns[index+1].attrs['y'] =
               node.parent.attrs['y']
            ns[index+1].attrs['z'] =
               node.parent.attrs['z']

## caso DIVIDE.
elif node.parent.type == 'divide':
    if (node.left_sibling == None):
        node.right_sibling.attrs['x']
           = node.attrs['x']+0.6*node.
           attrs['z']
        node.right_sibling.attrs['y']
           = node.attrs['y']
        node.right_sibling.attrs['z']
```

```
            = node.attrs['z']

        elif (node.right_sibling != None):
            node.right_sibling.attrs['x']
                = node.attrs['x']+0.6*node.
                attrs['z']
            node.right_sibling.attrs['y']
                = node.attrs['y']
            node.right_sibling.attrs['z']
                = node.attrs['z']

        ## caso nodo BARRA: encargado de
            posicionar correctamente el
            dividiendo,
        ## divisor y la barra de division.
        else:
            long_a = node.left_sibling.
                attrs['x'] - node.
                first_sibling().attrs['x']
            long_b = node.attrs['x'] -
                node.left_sibling.attrs['x'
                ]

            inicio_a = node.first_sibling
                ().attrs['x']
            fin_a = node.left_sibling.
                attrs['x']

            # movemos el denominador en el
                 eje x e y segun las
                expresiones
            # que haya

            if node.left_sibling.
                branch_has_type(['p', 'pu',
                 'divide']):
                 node.left_sibling.move(-
                    long_a,node.attrs['z'
                    ]+0.20)
            else:
                 node.left_sibling.move(-
                    long_a,node.attrs['z'
                    ]-0.09)
```

```
                inicio_b = node.left_sibling.
                    attrs['x']
                fin_b = node.attrs['x']-long_a

                node.attrs['x1'] = node.
                    first_sibling().attrs['x']

                # centrar B
                if (long_a > long_b):
                    node.attrs['x2'] = fin_a
                    node.left_sibling.move(((
                        long_a-long_b)/2.0),0)
                # centrar A
                elif (long_a < long_b):
                    node.attrs['x2'] = fin_b
                    node.first_sibling().move
                        (((long_b-long_a)/2.0)
                        ,0)
                else:
                    node.attrs['x2'] = fin_b

                if (index+1<len(ns)):
                    ns[index+1].attrs['x'] =
                        node.attrs['x2']
                    ns[index+1].attrs['y'] =
                        node.first_sibling().
                        attrs['y']-0.19
                    ns[index+1].attrs['z'] =
                        node.attrs['z']

            ## caso LLAVES.
            elif node.parent.type == 'brackets':
                ## caso '{': copia la informacion
                    al nodo derecho.
                if (node.left_sibling == None):
                    node.right_sibling.
                        copy_node_attrs(node)
                ## caso E:  avanza la variable 'x'
                elif (node.right_sibling != None):
                    node.right_sibling.attrs['x']
                        = node.attrs['x'] +0.6*node
                        .attrs['z']
                    node.right_sibling.attrs['y']
                        = node.attrs['y']
```

```
                node.right_sibling.attrs['z']
                    = node.attrs['z']
            ## caso '}': reestablece la
                configuracion de '{' para
            ## el siguiente nodo en el
                PREORDER.
            else:
                if (index+1<len(ns)):
                    if ns[index+1].type!='
                        brackets' or node.
                        parent.parent==None or
                        (node.parent.parent.
                        type != 'p' and node.
                        parent.parent.type != '
                        u' and node.parent.
                        parent.type != 'pu'):
                        ns[index+1].attrs['x']
                            = node.attrs['x']
                        if node.parent.
                            left_sibling!=None:
                            ns[index+1].attrs[
                                'y'] = node.
                                parent.
                                left_sibling.
                                attrs['y']
                            ns[index+1].attrs[
                                'z'] = node.
                                parent.
                                left_sibling.
                                attrs['z']
                        else:
                            ns[index+1].attrs[
                                'y'] = node.
                                attrs['y']
                            ns[index+1].attrs[
                                'z'] = node.
                                attrs['z']

        ## caso PARENTESIS
        elif node.parent.type == 'parens':
            ## caso '(': copia la informacion
                al nodo derecho avanzando 'x'
            if (node.left_sibling == None):
                node.right_sibling.attrs['x']
```

```
            = node.attrs['x'] +0.6*node
               .attrs['z']
            node.right_sibling.attrs['y']
               = node.attrs['y']
            node.right_sibling.attrs['z']
               = node.attrs['z']
        ## caso E:  avanza la variable 'x'
        elif (node.right_sibling != None):
            node.right_sibling.attrs['x']
               = node.attrs['x'] +0.6*node
               .attrs['z']
            node.right_sibling.attrs['y']
               = node.attrs['y']
            node.right_sibling.attrs['z']
               = node.attrs['z']
        ## caso ')': estira el '(' y
           reestablece la configuracion de
            ')' para
        ## el siguiente nodo en el
           PREORDER.
        else:
            h_l_attrs = node.left_sibling.
               find_higher_and_lower_attrs
               ()
            node.first_sibling().attrs['y1
               '] = h_l_attrs['y'][0]
            node.first_sibling().attrs['y2
               '] = h_l_attrs['y'][1]
            node.attrs['y1'] = h_l_attrs['
               y'][0]
            node.attrs['y2'] = h_l_attrs['
               y'][1]
            node.attrs['y'] = node.
               first_sibling().attrs['y']
            y_paren = (float(node.attrs['
               y2'])-float(node.attrs['y1'
               ]))*0.3
            if (index+1<len(ns)):
                if (index+1<len(ns)):
                    if ns[index+1].type!='
                       parens' or node.
                       parent.parent==None
                        or (node.parent.
                       parent.type != 'p'
```

```
                                        and node.parent.
                                        parent.type != 'u'
                                        and node.parent.
                                        parent.type != 'pu'
                                        ):
                                         ns[index+1].attrs[
                                             'x'] = node.
                                             attrs['x']+0.6*
                                             node.attrs['z']
                                         ns[index+1].attrs[
                                             'y'] = y_paren
                                         if node.parent.
                                             left_sibling!=
                                             None:
                                             ns[index+1].
                                                 attrs['z']
                                                 = node.
                                                 parent.
                                                 left_sibling
                                                 .attrs['z']
                                         else:
                                             ns[index+1].
                                                 attrs['z']
                                                 = node.
                                                 attrs['z']
                    index += 1
```

6.3. Apéndice III: SVG Builder - svg.py

```
import xml.etree.ElementTree as ET
from xml.etree.ElementTree import Element, SubElement,
    tostring

class SVG:

    def __init__(self):
        root = Element('svg')
        root.set('xmlns', 'http://www.w3.org/2000/svg'
            )
        root.set('version', '1.1')
        child = SubElement(root, 'g')
        child.set('transform', 'translate(0,300)_scale
            (200)')
        child.set('font-family', 'Courier')
```

```
        self.tree = ET.ElementTree(root)
        self.gElement = child

    def appendText(self, char, x, y, fontSize,
       translateX=0, translateY=0, scaleX=0, scaleY=0)
       :
        text = SubElement(self.gElement, 'text')
        text.text = char
        text.set('x', x)
        text.set('y', y)
        text.set('font-size', fontSize)
        if (translateX != 0 and translateY != 0 and
           scaleX != 0 and scaleY !=0):
            text.set('transform', 'translate('+
               translateX+','+translateY+') scale('+
               scaleX+','+scaleY+')')

    def appendLine(self,x1,y1,x2,y2):
        line = SubElement(self.gElement, 'line')
        line.set('x1',x1)
        line.set('x2',x2)
        line.set('y1',y1)
        line.set('y2',y2)
        line.set('stroke-width','0.03')
        line.set('stroke', 'black')

    def toString(self):
        root = self.tree.getroot()
        return tostring(root)

    def save(self, name):
        self.tree.write(name, xml_declaration=True)

class SVGBuilder:

    '''
    Clase encargada de armar el XML para el SVG a
       partir de un AST.
    '''

    def build(self, tree):
        svg = SVG()
        ns = tree.preorder_traversal()
        for node in ns:
```

```
            if (node.type == 'barra'):
                svg.appendLine(str(node.attrs['x1']),
                    str(node.attrs['y1']), str(node.
                    attrs['x2']), str(node.attrs['y2'])
                    )
            elif (node.type == '(' or node.type == ')'
                ):
                largo_paren = float(node.attrs['y2'])-
                    float(node.attrs['y1'])+float(node.
                    attrs['z'])
                y = largo_paren-float(node.attrs['z'])
                if node.type == '(':
                    svg.appendText('(', '0', str(node.
                        attrs['y']), str(node.attrs['z'
                        ]), str(node.attrs['x']), str(y
                        - 0.4*y), str(node.attrs['z'])
                        , str(largo_paren))
                else:
                    svg.appendText(')', '0', str(node.
                        attrs['y']), str(node.attrs['z'
                        ]), str(node.attrs['x']), str(y
                        - 0.4*y), str(node.attrs['z'])
                        , str(largo_paren))
            elif (node.type != 'brackets' and node.
                type != 'parens' and node.type != '{'
                and node.type != '}' \
                and node.type != 'concat' and node.
                    type != 'divide' and node.type != '
                    p' and node.type != 'u' \
                and node.type != 'pu'):
                svg.appendText(node.type, str(node.
                    attrs['x']), str(node.attrs['y']),
                    str(node.attrs['z']))
        return svg
```

6.4. Apéndice IV: Tests - test.py

```
import unittest
import sys
sys.path.insert(0,"../tree")
sys.path.insert(0,"../formula")

from tree import Tree
from tree import Node
```

```
from form import Form

class TestParser(unittest.TestCase):
    '''
    Testeando que los arboles AST sean generados
    correctamente a partir de varias expresiones.
    '''

    @classmethod
    def setUpClass(self):
        self.form = Form()

    def test_concat(self):
        exp = "ABCD"
        ast = self.form.parse(exp)
        ns = ast.root.preorder()
        res_actual = [n.type for n in ns]
        res_expected = ["concat", "A", "concat", "B",
            "concat", "C", "D"]
        self.assertEqual(res_actual, res_expected)

    def test_divide(self):
        exp = "A/B"
        ast = self.form.parse(exp)
        ns = ast.root.preorder()
        res_actual = [n.type for n in ns]
        res_expected = ["divide", "A", "B", "barra"]
        self.assertEqual(res_actual, res_expected)

    def test_superindice(self):
        exp = "A^B"
        ast = self.form.parse(exp)
        ns = ast.root.preorder()
        res_actual = [n.type for n in ns]
        res_expected = ["p", "A", "B"]
        self.assertEqual(res_actual, res_expected)

    def test_subindice(self):
        exp = "A_B"
        ast = self.form.parse(exp)
        ns = ast.root.preorder()
        res_actual = [n.type for n in ns]
        res_expected = ["u", "A", "B"]
        self.assertEqual(res_actual, res_expected)
```

```
def test_parentesis1(self):
    exp = "(A)"
    ast = self.form.parse(exp)
    ns = ast.root.preorder()
    res_actual = [n.type for n in ns]
    res_expected = ["parens", "(", "A", ")"]
    self.assertEqual(res_actual, res_expected)

def test_llaves(self):
    exp = "{A/B}C"
    ast = self.form.parse(exp)
    ns = ast.root.preorder()
    res_actual = [n.type for n in ns]
    res_expected = ['concat', 'brackets', '{', '
       divide', 'A', 'B', 'barra', '}', 'C']
    self.assertEqual(res_actual, res_expected)

def test_parentesis2(self):
    exp = "(A/B)C"
    ast = self.form.parse(exp)
    ns = ast.root.preorder()
    res_actual = [n.type for n in ns]
    res_expected = ['concat', 'parens', '(', '
       divide', 'A', 'B', 'barra', ')', 'C']
    self.assertEqual(res_actual, res_expected)

def test_super_sub_indice(self):
    exp = "A^B_C"
    ast = self.form.parse(exp)
    ns = ast.root.preorder()
    res_actual = [n.type for n in ns]
    res_expected = ["pu", "A", "B", "C"]
    self.assertEqual(res_actual, res_expected)

def test_combinado1(self):
    exp = "(A^BC^D/E^F_G+H)-I"
    ast = self.form.parse(exp)
    ns = ast.root.preorder()
    res_actual = [n.type for n in ns]
    res_expected = ["concat", "parens", "(", "
       divide", "concat", "p", "A", "B", "p", "C",
        "D", "concat", "pu", "E", "F", "G", "
       concat", "+", "H", "barra", ")", "concat",
```

```
            "_", "I"]
        self.assertEqual(res_actual, res_expected)

    def test_combinado2(self):
        exp = "(A^{BC})"
        ast = self.form.parse(exp)
        ns = ast.root.preorder()
        res_actual = [n.type for n in ns]
        res_expected = ['parens', '(', 'p', 'A', '
            brackets', '{', 'concat', 'B', 'C', '}', ')
            ']
        self.assertEqual(res_actual, res_expected)

    def test_combinado3(self):
        exp = "({A/B/C}{D/E})^{J+K}"
        ast = self.form.parse(exp)
        ns = ast.root.preorder()
        res_actual = [n.type for n in ns]
        res_expected = ['p', 'parens','(', 'concat', '
            brackets', '{', 'divide', 'A', 'divide', 'B
            ', 'C', 'barra', 'barra', '}', 'brackets',
            '{', 'divide', 'D', 'E', 'barra', '}', ')',
             'brackets', '{', 'concat', 'J', 'concat',
            '+', 'K', '}']
        self.assertEqual(res_actual, res_expected)

    def test_expresion_invalida1(self):
        exp = "^B"
        self.assertRaises(Exception, lambda: self.form
            .parse(exp))

    def test_expresion_invalida2(self):
        exp = "A^B^C"
        self.assertRaises(Exception, lambda: self.form
            .parse(exp))

    def test_expresion_invalida3(self):
        exp = "A_B_C"
        self.assertRaises(Exception, lambda: self.form
            .parse(exp))

if __name__ == '__main__':
    unittest.main()
```