


Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

# Trabajo Práctico 1 - Reentrega

## Teoria de Lenguajes

Segundo Cuatrimestre de 2015



| Apellido y Nombre | LU | E-mail |
|---|---|---|
| Rodriguez Pedro | 197/12 | pedro3110.jim@gmail.om |
| Matias Pizzagali | 257/12 | matipizza@gmail.com |

# Índice



# 1. Correcciones realizadas

Para la reentrega del TP, realizamos las siguientes modificaciones respecto de la primera entrega:

- Rehicimos el código, utilizando clases, aplicando “ buenas prácticas ” de programación
- Mejoramos la explicación sobre cómo realizamos el TP; agregamos al informe la TDS que generamos para resolver el problema propuesto
- Añadimos casos de test exitosos y fallidos

# 2. Introducción

El objetivo de este trabajo práctico es desarrollar un compositor de fórmulas matemáticas. El mismo tomará como entrada la descripción de una fórmula en una versión muy simplificada del lenguaje utilizado por LATEX y producirá como salida un archivo SVG (Scalable Vector Graphics).

# 3. Desarrollo

Para poder realizar el TP, utilizamos la librería PLY para Python, la cual implementa Lex y Yacc enteramente en Python. Este analizador léxico y sintáctico nos permitirá aceptar o rechazar cadenas de entrada de acuerdo a la gramática que definamos.

Una de las cosas que tuvimos que hacer para poder utilizar esta herramienta fue definir para este problema en particular los siguientes tokens, con sus respectivas expresiones regulares:

- ID : para cualquier caracter distinto a _, ^, (, ), { y }
- SUBINDEX : para la ER " / "
- SUPERINDEX : para la ER " ^ "
- DIVIDE : para la ER " /"
- LPAREN : para la ER " ( "
- RPAREN : para la ER " ) "
- LBRACKET : para la ER " { "
- RBRACKET : para la ER " } "

## 3.1. Desambiguación de la gramática

$$
\begin{array}{lcl}
E & \to & E\ E \\
 & | & E\ /\ E \\
 & | & E\ \hat{}\ E \\
 & | & E\ \_\ E \\
 & | & E\ \hat{}\ E\ \_\ E \\
 & | & E\ \_\ E\ \hat{}\ E \\
 & | & (\ E\ ) \\
 & | & \{\ E\ \} \\
 & | & l
\end{array}
\qquad
\begin{array}{lcl}
S & \to & E \\
E & \to & E\ /\ A \\
 & | & A \\
A & \to & A\ B \\
 & | & B \\
B & \to & C \\
 & | & C\ \hat{}\ C \\
 & | & C\ \_\ C \\
 & | & C\ \hat{}\ C\ \_\ C \\
 & | & C\ \_\ C\ \hat{}\ C \\
C & \to & (\ E\ ) \\
 & | & \{\ E\ \} \\
 & | & l
\end{array}
$$

La gramática propuesta inicialmente fue la de la izquierda. Sin embargo, al utilizar esta gramática para hacer el parsing, se generan conflictos Shift/Reduce. Esto es debido a que la gramática es ambigua (para una misma cadena pueden haber más de un árbol de derivación posibles). Luego, para evitar conflictos Shift/Reduce, generamos la otra gramática expuesta, equivalente a la anterior pero con la diferencia de que no es ambigua.

Tuvimos en cuenta que la división es la operación de menor precedencia, seguida de la concatenación, luego los subíndice y superíndice y finalmente los paréntesis y corchetes. También que división y concatenación son asociativas a izquierda y que subíndice y superíndice no son asociativos.

## 3.2. TDS y Atributos

Para cada uno de los símbolos S,A,B,C,E (no terminales) y ID (terminal) definimos los siguientes atributos heredados de tipo float:

- $x$ : para guardar el comienzo de cada subexpresión sobre el eje x
- $y$ : para guardar el comienzo de cada subexpresión sobre el eje y
- $tam$ : para guardar el tamaño (escala) de cada subexpresión

Y los siguientes atributos sintetizados, también de tipo float.

- $ancho$ : para guardar el ancho de cada subexpresión
- $h\_up$ : considerando que cada expresión esta encapsuladad en un rectángulo, este atributo contiene la distancia que hay entre la línea sobre la que se escriben los caracteres y el techo del rectángulo
- $h\_down$ : igual que $h\_up$, con la diferencia que contiene la distancia que hay entre la línea sobre la que se escriben los caracteres y la base del rectángulo
- $altura$ : contiene la altura total de la subexpresión. Es la suma de $h\_up$ y $h\_down$, y agregamos este atributo sólo por comodidad

A partir de estos atributos, generamos la siguiente TDS:

- $S \to \{E.tam = 1, E.x = 0, E.y = 0\}\mathbf{E}$
- $E_1 \to \{E_2.tam = E_1.tam, E_2.x = E_1.x + E_1.ancho/2 - E_2.ancho/2, E_2.y = E_1.y - E_2.h\_down - 0{,}2 * E_1.tam\}\mathbf{E}_2/$
  $\{A.tam = E_1.tam, A.x = E_1.x + E_1.ancho/2 - A.ancho/2, A.y = E_1.y + A.h\_up + 0{,}1 * E_1.tam\}\mathbf{A}$
  $\{E_1.h\_up = E_2.altura + 0{,}35 * E_1.tam, E_1.h\_down = A.altura + 0{,}3 * E_1.tam, E_1.ancho = max(E_2.ancho, A.ancho), E_1.altura = E_1.h\_up + E_1.h\_down\}$
- $E \to \{A.tam = E.tam, A.x = E.x, A.y = E.y\}\mathbf{A}$
- $A_1 \to \{A_2.tam = A_1.tam, A_2.x = A_1.x, A_2.y = A_1.y\}\mathbf{A}_2$
  $\{B.tam = 1, B.x = A_2.x + A_2.ancho, B.y = A_1.y\}\mathbf{B}$
  $\{A_1.h\_up = max(A_2.h\_up, B.h\_up), A_1.h\_down = max(A_2.h\_down, B.h\_down), A_1.ancho = A_2.ancho + B.ancho, A_1.altura = A_1.h\_up + A_1.h\_down\}$
- $A \to \{B.tam = 1, B.x = A.x, B.y = A.y\}\mathbf{B}$
  $\{A.h\_up = B.h\_up, A.h\_down = B.h\_down, A.altura = B.altura, A.ancho = B.ancho\}$
- $B \to \{C.tam = B.tam, C.x = B.x, C.y = B.y\}\mathbf{C}$
  $\{B.ancho = C.ancho, B.h\_up = C.h\_up, B.h\_down = C.h\_down\}$
- $B \to \{C_1.tam = B.tam, C_1.x = B.x, C_1.y = B.y\}\mathbf{C}_1\hat{}$
  $\{C_2.tam = C_1.tam * 0{,}7, C_2.x = C_1.x + C_1.ancho, C_2.y = C_1.y - C_2.h\_down - 0{,}5 * C_1.tam\}\mathbf{C}_2$
  $\{B.ancho = C_1.ancho + C_2.ancho, B.h\_up = C_1.h\_up + C_2.altura * 0{,}7, B.h\_down = C_1.h\_down B.altura = B.h\_down + B.h\_up\}$
- $B \to \{C_1.tam = B.tam, C_1.x = B.x, C_1.y = B.y\}\mathbf{C}_1\_$
  $\{C_2.tam = C_1.tam * 0{,}7, C_2.x = C_1.x + C_1.ancho, C_2.y = C_1.y + C_2.h\_up\}\mathbf{C}_2$

$\{B.ancho = C_1.ancho + C_2.ancho, B.h\_up = C_1.h\_up, B.h\_down = C_1.h\_down + 0{,}7 * C_2.altura, B.altura = B.h\_up + B.h\_down\}$

- $B \rightarrow \{C_1.tam = B.tam, C_1.x = B.x, C_1.y = B.y\}\mathbf{C_1}\,\hat{}$
  $\{C_2.tam = C_1.tam * 0{,}7, C_2.x = C_1.x + C_1.ancho, C_2.y = C_1.y - C_2.h\_down - 0{,}5 * C_1.tam\}\mathbf{C_2}\_$
  $\{C_3.tam = C_1.tam * 0{,}7, C_3.x = C_1.x + C_1.ancho, C_3.y = C_1.y + C_3.h\_up\}\mathbf{C_3}$
  $\{B.ancho = C_1.ancho + max(C_2.ancho, C_2.ancho), B.h\_up = C_1.h\_up + C_2.altura * 0{,}7, B.h\_down = C_1.h\_down + C_3.altura * 0{,}7, B.altura = B.h\_up + B.h\_down\}$

- $B \rightarrow \{C_1.tam = B.tam, C_1.x = B.x, C_1.y = B.y\}\mathbf{C_1}\_$
  $\{C_2.tam = C_1.tam * 0{,}7, C_2.x = C_1.x + C_1.ancho, C_2.y = C_1.y + C_2.h\_up\}\mathbf{C_2}\,\hat{}$
  $\{C_3.tam = C_1.tam * 0{,}7, C_3.x = C_1.x + C_1.ancho, C_3.y = C_1.y - C_3.h\_down - 0{,}5 * C_1.tam\}\mathbf{C_3}$
  $\{B.ancho = C_1.ancho + max(C_2.ancho, C_2.ancho), B.h\_up = C_1.h\_up + C_3.altura * 0{,}7, B.h\_down = C_1.h\_down + C_2.altura * 0{,}7, B.altura = B.h\_up + B.h\_down\}$

- $C \rightarrow (\{E.x = C.x + C.tam * 0{,}3, E.y = C.y, E.tam = C.tam\}\mathbf{E})$
  $\{C.h\_up = E.h\_up, C.h\_down = E.h\_down, C.ancho = E.ancho + 2 * C.tam * 0{,}3, C.altura = C.h\_down + C.h\_up\}$

- $C \rightarrow \mathbf{\{}\{E.x = C.x, E.y = C.y, E.tam = C.tam\}\mathbf{E\}}$
  $\{C.h\_up = E.h\_up, C.h\_down = E.h\_down, C.ancho = E.ancho, C.altura = E.altura\}$

- $C \rightarrow l\{C.h\_up = 0{,}8 * C.tam, C.h\_down = 0{,}2 * C.tam, C.altura = C.\_up + C.h\_down, C.ancho = 0{,}6 * C.tam\}$

Para implementar esta TDS, generamos para cada cadena de entrada un árbol sintáctico en el cual cada hoja está dada por el símbolo terminal $l$. Llamamos al nodo-hoja que va a caracterizar a este símbolo terminal ID.

Para trabajar con el resto de los símbolos (no terminales), definimos los siguientes nodos:

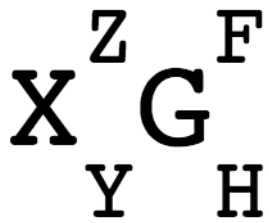
- ROOT: es el nodo raíz. Allí inicializamos el tamaño inicial de la expresión, y calculamos la posición $y$ de la expresión para que posteriormente quede centrada en el espacio donde la dibujamos. Tiene 1 hijo.
- (): para caracterizar a la subexpresión contenida entre las ER " ( " y " ) ". Cuenta con 1 hijo
- CONCAT: para caracterizar la concatenación de dos subexpresiones válidas. Cuenta con 2 hijos
- SUPERINDEX: para caracterizar dos subexpresiones unidas por la ER ^. Cuenta con 2 hijos
- SUBINDEX: para caracterizar dos subexpresiones unidas por la ER _. Cuenta con 2 hijos
- SUBSUPERINDEX: para caracterizar tres subexpresiones unidas las primeras dos por un _y la segunda y tercera por un ^. Cuenta con 3 hijos
- SUPERSUBINDEX: para caracterizar tres subexpresiones unidas las primeras dos por un ^ y las segunda y tercera por un _. Cuenta con 3 hijos
- DIVISION: para caracterizar dos subexpresiones unidas mediante la ER de división, " / ". Cuenta con 2 hijos

Cada nodo y cada hoja tiene los mismos 7 atributos que tienen los símbolos de nuestra gramática. Para heredar y sintetizar de forma correcta dichos atributos según la TDS ya descripta, recorremos este árbol 3 veces (la primera y última de forma top-down y la segunda bottom-up). Esto lo hacemos en los siguientes 3 recorridos (etapas):

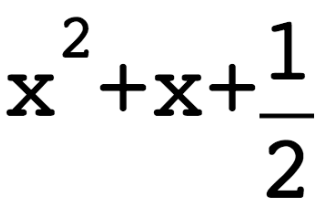
- recorrer: heredamos desde la raíz hasta las hojas el atributo *tam*
- recorrer2: sintetizamos desde las hojas hasta la raíz los atributos *altura*, *ancho*, *h_up* y *h_down*, utilizando el atributo *tam*, ya calculado.
- dump_ast: heredamos desde la raíz hasta las hojas los atributos $x$ e $y$, utilizando los atributos ya calculados en los dos recorridos anteriores. Como en esta etapa obtenemos el árbol decorado correctamente con todos los atributos, también aquí generamos el archivo SVG buscado.

## 4. Resultados y casos de prueba

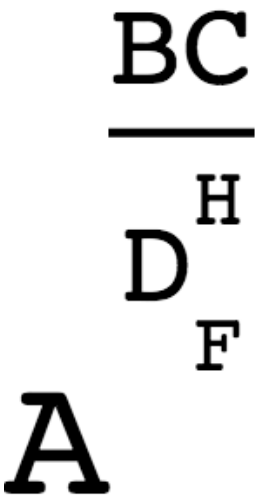
A continuación, exponemos una serie de figuras, cada una obtenida a partir de la cadena que nombra a dicha figura.



X_YˆZGˆF_H

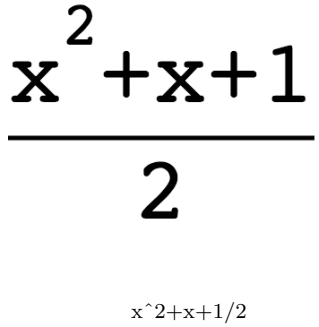


xˆ2+x+{1/2}

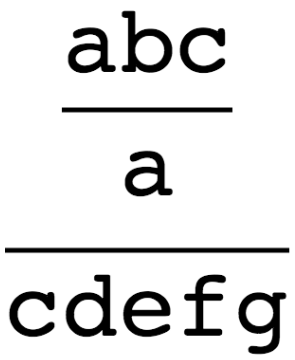


Aˆ{BC/D_FˆH}

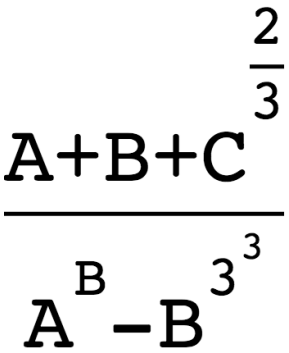
(a)



xˆ2+x+1/2

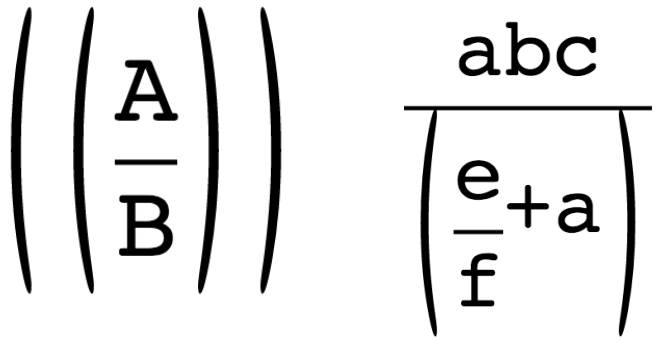


abc/a/cdefg



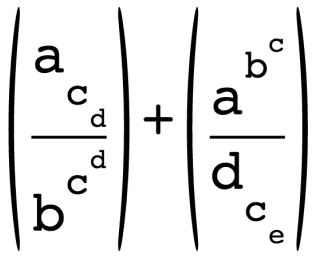
A+B+Cˆ{2/3}/{Aˆ{B}-Bˆ{3ˆ3}}

(b)



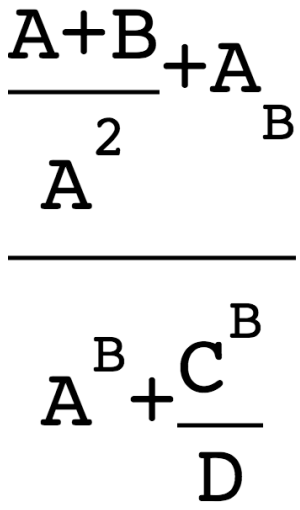
((A/B))

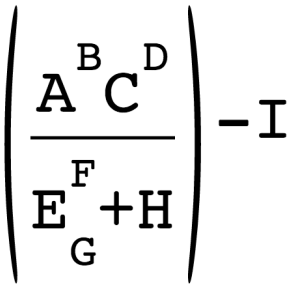
abc/({e/f}+a)

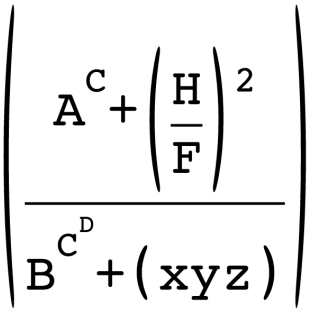


({a_{c_d}/b^{c^d}})+({a^{b^c}/d_{c_e}})

(c)







{A+B/A^2}+A_B/{A^B+{C^B/D}

(A^BC^D/E^F_G+H)-I

(A^C+(H/F)^2/B^C^D+(xyz))

(d)

A continuación exponemos algunos ejemplos de cadenas que no son aceptadas por el programa (debido a que no son aceptadas por la gramática usada):

| | |
|---|---|
| {A | Error: Cadena no valida |
| A^B^C | Error: Cadena no valida |
| A_B_C | Error: Cadena no valida |
| A_(B_)C | Error: Cadena no valida |

| | |
|---|---|
| A//B | Error: Cadena no valida |
| () | Error: Cadena no valida |
| {(}) | Error: Cadena no valida |
| {(AB}AB) | Error: Cadena no valida |

# 5. Manual de usuario

Para correr el TP, es necesario tener instalado Python 2.7 o posterior y la librería PLY, que puede obtenerse en https://pypi.python.org/pypi/ply. Para correr el tp, abrir una terminal en la carpeta src, y ejecutar el comando " python tp.py ". A continuación, introducir la cadena que se desea generar. Se genera la figura deseada con la extensión " .svg " en el archivo " figura.svg ".

# 6. Conclusiones

La realización de este Trabajo Práctico fue interesante, ya que nos permitió generar una aplicación concreta de toda la teoría vista en la segunda parte de la materia, básicamente sobre gramáticas de atributos y traducción dirigida por sintaxis (TDS). Entendimos mejor la relación que hay entre un árbol sintáctico, una gramática y una TDS para la generación de una herramienta de parsing de utilidad en la vida real.

Como se puede ver en los casos de test, las figuras que genera nuestro programa son de un nivel de calidad aceptable ya que en todos los casos expuestos se nota muy bien cuál es la expresión lógica-matemática que se pretende generar. Obviamente la herramienta podría ser mejorada para que las figuras queden " más lindas ", pero por lo que vimos en el desarrollo del TP, esta no es una tarea fácil ya que para hacer esto hay que hacer mucha experimentación y fijar una gran variedad de parámetros y constantes para considerar la gran variedad de combinaciones que se pueden generar a partir los símbolos y la gramática considerada.

# 7. Codigo

## 7.1. tp.py

```
# -----------------------------------------------------------------------------
# tp.py
#
# -----------------------------------------------------------------------------
import lexer_rules
import parser_rules
import pdb

from sys import argv, exit

import ply.lex as lex
import ply.yacc as yacc

if __name__ == "__main__":
  # Build the lexer
  lexer = lex.lex(module=lexer_rules)
  # Build the parser
  parser = yacc.yacc(module=parser_rules)

  s = "init"
  while s != "exit()": # finalizo si en la entrada escriben exit() - asumimos
      que es una cadena no valida (?)
    try:
      s = raw_input('cadena > ')
    except EOFError:
      break
    if not s: continue
    ast = parser.parse(s, lexer)

    ast.recorrer()
    ast.recorrer2()
```

```
    out = []
    ast.dump_ast(out)

    open('fig.svg','w').write(''.join(out))

    print out
```

## 7.2. parser_rules.py

```
from lexer_rules import tokens

from expressions import *

import pprint

def p_expression_init(subexpressions):
  'S : E'
  subexpressions[0] = Root(subexpressions[1])

# E -> E / A | A
def p_expression_E1(subexpressions):
  'E : E DIVIDE A'
  subexpressions[0] = Divide(subexpressions[1], subexpressions[3])

def p_expression_E2(subexpressions):
  'E : A'
  subexpressions[0] = subexpressions[1]

# A -> A B | B
def p_expression_A1(subexpressions):
  'A : A B'
  subexpressions[0] = Concat(subexpressions[1], subexpressions[2])

def p_expression_A2(subexpressions):
  'A : B'
  subexpressions[0] = subexpressions[1]

# B -> C | C SUPERINDEX C | C SUBINDEX C | C SUPERINDEX C SUBINDEX C | C
    SUBINDEX C SUPERINDEX C
def p_expression_B1(subexpressions):
  'B : C'
  subexpressions[0] = subexpressions[1]

def p_expressionB2(subexpressions):
  'B : C SUPERINDEX C'
  subexpressions[0] = SuperIndex(subexpressions[1], subexpressions[3])

def p_expressionB3(subexpressions):
  'B : C SUBINDEX C'
  subexpressions[0] = SubIndex(subexpressions[1], subexpressions[3])

def p_expressionB4(subexpressions):
  'B : C SUPERINDEX C SUBINDEX C'
  subexpressions[0] = SuperSubIndex(subexpressions[1], subexpressions[3],
      subexpressions[5])

def p_expressionB5(subexpressions):
  'B : C SUBINDEX C SUPERINDEX C'
  subexpressions[0] = SubSuperIndex(subexpressions[1], subexpressions[3],
      subexpressions[5])

# C -> { E }
```

```
def p_expression_C1(subexpressions):
  'C : LBRACKET E RBRACKET'
  subexpressions[0] = subexpressions[2]

# C -> ( E )
def p_expression_C2(subexpressions):
  'C : LPAREN E RPAREN'
  subexpressions[0] = Parenthesis(subexpressions[2])

# C -> ID
# tengo que escribir el valor del token ID
def p_expression_C3(subexpressions):
  'C : ID'
  subexpressions[0] = Id(subexpressions[1])

# Error rule for syntax errors
def p_error(p):
  raise Exception("Syntax error.")
```

### 7.3. lexer_rules.py

```
# Lista de tokens
tokens = [
    'ID',           # es hoja
    'DIVIDE',       # es nodo: /
    'LPAREN',       # es nodo: (
    'RPAREN',       # es nodo: )
    'LBRACKET',     # para separar expresiones sin agregar ningun simbolo
    'RBRACKET',     # para separar expresiones sin agregar ningun simbolo
    'SUPERINDEX',   # es nodo: ^
    'SUBINDEX'     # es nodo: _
]

# Expresiones regulares para cada token
t_ignore  = ' \t' # ignoramos los espacios
t_DIVIDE  = r'/'
t_LPAREN  = r'\('
t_RPAREN  = r'\)'
t_LBRACKET = r'\{'
t_RBRACKET = r'\}'
t_SUPERINDEX = r'\^'
t_SUBINDEX = r'\_'

def t_ID(t):
    r'[a-zA-Z0-9+-]'
    t.value = t.value
    return t


def t_error(t):
    message = "Token desconocido:"
    message += "\ntype:" + t.type
    message += "\nvalue:" + str(t.value)
    message += "\nline:" + str(t.lineno)
    message += "\nposition:" + str(t.lexpos)
    raise Exception(message)
```

### 7.4. expressions.py

```
ANCHO_CARACTER_DEFAULT = 0.6

def initialize_atts(obj):
  obj.tam = 0
```

```
  obj.ancho = 0
  obj.altura = 0
  obj.x = 0
  obj.y = 0
  obj.h_up = 0
  obj.h_down = 0

def make_text(x,y,tam,char):
  return '''<text x="''' + x + '''" y="''' + y + '''" font-size="''' + tam +
      '''">''' + char + '''</text>'''

def make_line(x1,y1,x2,y2,ancho):
  return '''<line x1="''' + x1 + '''" y1="''' + y1 + '''" x2="''' + x2 + '''"
      y2="''' + y2 + '''" stroke-width="''' + ancho + '''" stroke="black"/>'''

def make_open_paren(x,y,tam,t1,t2,s1,s2):
  return '''<text x="''' + x + '''" y="''' + y + '''" font-size="''' + tam +
      '''" transform="translate(''' + t1 + ''','''' + t2 + ''') scale(''' + s1 +
      ''','''' + s2 + ''')">(</text>'''

def make_close_paren(x,y,tam,t1,t2,s1,s2):
  return '''<text x="''' + x + '''" y="''' + y + '''" font-size="''' + tam +
      '''" transform="translate(''' + t1 + ''','''' + t2 + ''') scale(''' + s1 +
      ''','''' + s2 + ''')">)</text>'''

class Root(object):
  def __init__(self, hijo):
    initialize_atts(self)
    self.hijo = hijo

  def name(self):
    return "Start"

  def children(self):
    return [self.hijo]

  def recorrer(self):
    self.hijo.tam = 1
    self.hijo.recorrer()

  def recorrer2(self):
    self.hijo.recorrer2()
    self.ancho = self.hijo.ancho
    self.h_up = self.hijo.h_up
    self.h_down = self.hijo.h_down
    self.altura = self.hijo.altura

  def dump_ast(self, out):
    self.hijo.x = 0
    self.hijo.y = self.h_up
    out.append ( '''<?xml version="1.0" standalone="no"?> <!DOCTYPE svg PUBLIC
        "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.
        dtd"> <svg xmlns="http://www.w3.org/2000/svg" version="1.1"> <g
        transform="translate(0, 50) scale(50)" font-family= "Courier">''' )

    self.hijo.dump_ast(out)

    out.append ( '''</g></svg> ''' )
```

```
class Divide(object):
  def __init__(self, numerador, denominador):
    initialize_atts(self)
    self.numerador = numerador
    self.denominador = denominador

  def name(self):
    return "Divide"

  def children(self):
    return [self.numerador, self.denominador]

  def recorrer(self):
    self.numerador.tam = self.tam
    self.numerador.recorrer()
    self.denominador.tam = self.tam
    self.denominador.recorrer()

  def recorrer2(self):
    self.numerador.recorrer2()
    self.denominador.recorrer2()
    self.ancho = max(self.denominador.ancho, self.numerador.ancho)
    self.h_up = self.numerador.altura + 0.40 * self.tam
    self.h_down = self.denominador.altura + 0.28 * self.tam
    self.altura = self.h_up + self.h_down

  def dump_ast(self, out):
    self.numerador.x = self.x + self.ancho/2 - self.numerador.ancho/2
    self.numerador.y = self.y - (self.numerador.h_down + 0.2 * self.tam)
    self.numerador.dump_ast(out)

    out.append( make_line(str(self.x), str(self.y), str(self.x + self.ancho),
        str(self.y), str(0.05) ) )

    self.denominador.x = self.x + self.ancho/2 - self.denominador.ancho/2
    self.denominador.y = self.y + (self.denominador.h_up + 0.1 * self.tam)
    self.denominador.dump_ast(out)




class Concat(object):
  def __init__(self, left, right):
    initialize_atts(self)
    self.left = left
    self.right = right

  def name(self):
    return "Concat"

  def children(self):
    return [self.left, self.right]

  def recorrer(self):
    self.left.tam = self.tam
    self.left.recorrer()
    self.right.tam = self.tam
    self.right.recorrer()

  def recorrer2(self):
    self.left.recorrer2()
    self.right.recorrer2()
```

```
    self.ancho = self.left.ancho + self.right.ancho

    self.h_up = max(self.left.h_up, self.right.h_up)
    self.h_down = max(self.left.h_down, self.right.h_down)
    self.altura = self.h_down + self.h_up

  def dump_ast(self, out):
    self.left.x = self.x
    self.left.y = self.y
    self.left.dump_ast(out)

    self.right.x = self.x + self.left.ancho
    self.right.y = self.y
    self.right.dump_ast(out)



class SuperIndex(object):
  def __init__(self, base, index):
    initialize_atts(self)
    self.base = base
    self.index = index

  def name(self):
    return "SuperIndex"

  def children(self):
    return [self.base, self.index]

  def recorrer(self):
    self.base.tam = self.tam
    self.base.recorrer()
    self.index.tam = self.tam * 0.7
    self.index.recorrer()

  def recorrer2(self):
    self.base.recorrer2()
    self.index.recorrer2()
    self.ancho = self.base.ancho + self.index.ancho

    self.h_up = self.base.h_up + self.index.altura * 0.7
    self.h_down = self.base.h_down
    self.altura = self.h_down + self.h_up

  def dump_ast(self, out):
    self.base.x = self.x
    self.base.y = self.y
    self.base.dump_ast(out)

    self.index.x = self.base.x + self.base.ancho
    self.index.y = self.y - (self.index.h_down + self.base.tam * 0.5)
    self.index.dump_ast(out)



class SubIndex(object):
  def __init__(self, base, index):
    initialize_atts(self)
    self.base = base
```

```
    self.index = index

  def name(self):
    return "SubIndex"

  def children(self):
    return [self.base, self.index]

  def recorrer(self):
    self.base.tam = self.tam
    self.base.recorrer()
    self.index.tam = self.tam * 0.7
    self.index.recorrer()

  def recorrer2(self):
    self.base.recorrer2()
    self.index.recorrer2()
    self.ancho = self.base.ancho + self.index.ancho

    self.h_up = self.base.h_up
    self.h_down = self.base.h_down + self.index.altura * 0.7
    self.altura = self.h_up + self.h_down

  def dump_ast(self, out):
    self.base.x = self.x
    self.base.y = self.y
    self.base.dump_ast(out)

    self.index.x = self.base.x + self.base.ancho
    self.index.y = self.base.y + (self.index.h_up)
    self.index.dump_ast(out)



class SuperSubIndex(object):
  def __init__(self, base, super_index, sub_index):
    initialize_atts(self)
    self.base = base
    self.super_index = super_index
    self.sub_index = sub_index

  def name(self):
    return "SuperSubIndex"

  def children(self):
    return [self.base, self.super_index, self.sub_index]

  def recorrer(self):
    self.base.tam = self.tam
    self.base.recorrer()
    self.super_index.tam = self.tam * 0.7
    self.super_index.recorrer()
    self.sub_index.tam = self.tam * 0.7
    self.sub_index.recorrer()

  def recorrer2(self):
    self.base.recorrer2()
    self.super_index.recorrer2()
    self.sub_index.recorrer2()
    self.ancho = self.base.ancho + max(self.super_index.ancho, self.sub_index.
        ancho)
```

```
    self.h_up = self.base.h_up + self.super_index.altura * 0.7
    self.h_down = self.base.h_down + self.sub_index.altura * 0.7
    self.altura = self.h_up + self.h_down

  def dump_ast(self, out):
    self.base.x = self.x
    self.base.y = self.y
    self.base.dump_ast(out)

    self.super_index.x = self.base.x + self.base.ancho
    self.super_index.y = self.base.y - (self.super_index.h_down + self.base.tam
        * 0.5)
    self.super_index.dump_ast(out)

    self.sub_index.x = self.base.x + self.base.ancho
    self.sub_index.y = self.base.y + (self.sub_index.h_up)
    self.sub_index.dump_ast(out)



class SubSuperIndex(object):
  def __init__(self, base, sub_index, super_index):
    initialize_atts(self)
    self.base = base
    self.sub_index = sub_index
    self.super_index = super_index

  def name(self):
    return "SubSuperIndex"

  def children(self):
    return [self.base, self.sub_index, self.super_index]

  def recorrer(self):
    self.base.tam = self.tam
    self.base.recorrer()
    self.sub_index.tam = self.tam * 0.7
    self.sub_index.recorrer()
    self.super_index.tam = self.tam * 0.7
    self.super_index.recorrer()

  def recorrer2(self):
    self.base.recorrer2()
    self.super_index.recorrer2()
    self.sub_index.recorrer2()
    self.ancho = self.base.ancho + max(self.super_index.ancho, self.sub_index.
        ancho)

    self.h_up = self.base.h_up + self.super_index.altura * 0.7
    self.h_down = self.base.h_down + self.sub_index.altura * 0.7
    self.altura = self.h_up + self.h_down

  def dump_ast(self, out):
    self.base.x = self.x
    self.base.y = self.y
    self.base.dump_ast(out)

    self.sub_index.x = self.base.x + self.base.ancho
    self.sub_index.y = self.base.y + (self.sub_index.h_up)
    self.sub_index.dump_ast(out)

    self.super_index.x = self.base.x + self.base.ancho
    self.super_index.y = self.base.y - (self.super_index.h_down + self.base.tam
```

```
        * 0.5)
    self.super_index.dump_ast(out)



class Parenthesis(object):
  def __init__(self, content):
    initialize_atts(self)
    self.content = content

  def name(self):
    return "Parenthesis"

  def children(self):
    return [self.content]

  def recorrer(self):
    self.content.tam = self.tam
    self.content.recorrer()

  def recorrer2(self):
    self.content.recorrer2()
    self.ancho = self.content.ancho + (2 * self.tam * ANCHO_CARACTER_DEFAULT)
    self.h_up = self.content.h_up
    self.h_down = self.content.h_down
    self.altura = self.h_down + self.h_up

  def dump_ast(self, out):
    self.content.x = self.x + self.tam * ANCHO_CARACTER_DEFAULT
    self.content.y = self.y

    out.append( make_open_paren(str(0), str(0), str(1), str(self.x),str(self.y
        + self.h_down - 0.25 * self.altura),
                                str(1), str(self.altura * 1.3) ) )
    self.content.dump_ast(out)
    out.append( make_close_paren( str(0), str(0), str(1), str(self.x + self.
        ancho - self.tam * ANCHO_CARACTER_DEFAULT), str(self.y + self.h_down -
        0.25 * self.altura),
                                str(1), str(self.altura * 1.3)) )


class Id(object):
  def __init__(self, value):
    initialize_atts(self)
    self.value = value

  def name(self):
    return "ID"

  def children(self):
    return []

  def recorrer(self):
    return

  def recorrer2(self):
    self.ancho = self.tam * ANCHO_CARACTER_DEFAULT # el ancho es 0.6 del tam
        del caracter
    self.h_up = 0.8 * self.tam
    self.h_down = 0.2 * self.tam
    self.altura = self.h_up + self.h_down
```

```
def dump_ast(self, out):
  out.append( make_text(str(self.x), str(self.y), str(self.tam), str(self.
      value) ) )
```