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

Trabajo Práctico 1

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. 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).

2. Desarrollo

2.1. Desambiguación de la gramática

Para poder realizar el TP, utilizamos la librería PLY para Python, la cual permite parsear las cadenas de entrada en función de una gramática que nosotros le proveemos.

La siguiente gramática ambigua fue la propuesta por la cátedra:

$$
\begin{array}{lcl}
E & \to & E\ E \\
 & | & E\ /\ E \\
 & | & E\ \hat{}\ E \\
 & | & E\ \_\ E \\
 & | & E\ \hat{}\ E\ \_\ E \\
 & | & E\ \_\ E\ \hat{}\ E \\
 & | & (\ E\ ) \\
 & | & \{\ E\ \} \\
 & | & l
\end{array}
$$

Como esta gramátia es ambigua, genera conflictos Shift/Reduce y Reduce/Reduce. Es por esto que lo primero que hicimos fue desambiguarla, para obtener la siguiente gramática alternativa, que genera el mismo lenguaje.

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

Tuvimos en cuenta que la división es la operación de menor precedencia, seguida de la concatenación. También que ambas son asociativas a izquierda y que el super y sub índice no son asociativos.

2.2. TDS y Atributos

Para cada uno de los nodos del árbol (CONCAT, DIVIDE, (), SUBINDEX, SUPERINDEX, SUBSUPERINDEX, SUPERSUBINDEX y ID) definimos los siguientes atributos: x, y, $tam(alto)$, $h1$, $h2$ y $a(ancho)$. Estos atributos son todos sintetizados.

Una vez que tenemos el árbol sintáctico, recorremos el mismo 3 veces, modificando los valores de los atributos. En el primer recorrido, definimos $tam = 1$ para el nodo raíz, y hacemos que cada padre defina para cada uno de sus hijos el atributo tam. Por ejemplo, se verifica que el tam de cada subíndice y superíndice es $0{,}7 * tam'$, donde tam' es el atributo tam de su padre.

En el segundo recorrido, definimos el ancho $'a'$ que ocupa cada nodo en función del ancho total que es ocupado entre todos sus descendientes.
Por último, hacemos otro recorrido top-down, en el cuál a partir de los atribtos $'tam'$ y $'a'$ definimos los atributos $'x'$ e $'y'$, que son los que van a determinar en qué posición se escribirá cada una de las subexpresiones de la cadena, y efectuamos la traducción a SVG.

3. Resultados

4. Codigo

4.1. tp.py

```
# -----------------------------------------------------------------------------
# tp.py
#
# -----------------------------------------------------------------------------


# ejemplo del tp: (A^BC^D/E^F_G+H)-I
'''
        <text x="0" y="0" font-size="1" transform=
              "translate(0, 1.36875) scale(1,2.475)">(</text>
        <text x=".69" y=".53" font-size="1">A</text>
        <text x="1.29" y=".08" font-size=".7">B</text>
        <text x="1.71" y=".53" font-size="1">C</text>
        <text x="2.31" y=".08" font-size=".7">D</text>
        <line x1="0.6" y1="0.72" x2="2.82" y2=".72"
              stroke-width="0.03" stroke="black"/>
        <text x="0.6" y="1.68" font-size="1">E</text>
        <text x="1.2" y="1.93" font-size=".7">G</text>
        <text x="1.2" y="1.23" font-size=".7">F</text>
        <text x="1.62" y="1.68" font-size="1">+</text>
        <text x="2.22" y="1.68" font-size="1">H</text>
        <text x="0" y="0" font-size="1" transform =
              "translate(2.82, 1.36875) scale(1,2.475)">)</text>
        <text x="3.42" y="1" font-size="1">-</text>
        <text x="4.02" y="1" font-size="1">I</text>
    '''

import lexer_rules
import parser_rules

from sys import argv, exit

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

# Build the lexer
lexer = lex.lex(module=lexer_rules)

# Build the parser
parser = yacc.yacc(module=parser_rules)

while True:
   try:
       s = raw_input('cadena > ')
   except EOFError:
       break
   if not s: continue
   result = parser.parse(s)
```

```
    print(result)
```

4.2. lexer_rules.py

```
# List of token names.   This is always required
tokens = (
   'ID',          # es hoja
   'DIVIDE',      # es nodo: /
   'LPAREN',
   'RPAREN',
   'LBRACKET',    # es nodo: ()
   'RBRACKET',
   'SUPERINDEX',  # es nodo: ^
   'SUBINDEX',     # es nodo: _
)

# Regular expression rules for simple tokens
t_ignore  = ' \t'
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-Z+-]'
    t.value = t.value
    return t

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

4.3. parser_rules.py

```
from lexer_rules import tokens

import pprint

# Mientras voy parseando, creo el arbol sintactico de la expresion.
nodos = { 'DIVIDE', 'CONCAT', 'SUPERINDEX', 'SUBINDEX',
  'SUBSUPERINDEX', 'SUPERSUBINDEX', 'ID', '()' }

# Los atributos con los que voy a ir decorando el arbol sintactico
atributosNil = { 'tam': None, 'x': None, 'y': None,
  'a': None, 'h1': None, 'h2': None}

def p_expression_init(p):
    'S : E'
    res1 = recorrer(p[1],1)      # agrego todos los atributos menos x e y
    res2 = recorrer2(res1,0.5,-0.7)  # agrego x e y

    pp = pprint.PrettyPrinter(indent=1)
    pp.pprint(res2)

    out = []
    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, 200) scale(200)" font-family= "Courier">''' )

    # recorro el arbol y voy agregando lineas al output segun corresponda
```

```
    recorrer3(res2, out)

    # fin
    out.append ( '''</g>
                    </svg> ''' )
    open('test.svg','w').write(''.join(out))
    print out

# E -> E / A | A
def p_expression_E1(p):
    'E : E DIVIDE A'
    p[0] = { 'DIVIDE': [ p[1], p[3] ], 'attr': atributosNil.copy() }

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

# A -> A B | B
def p_expression_A1(p):
    'A : A B'
    p[0] = { 'CONCAT': [ p[1], p[2] ], 'attr': atributosNil.copy() }

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

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

def p_expressionB2(p):
    'B : C SUPERINDEX C'
    p[0] = { 'SUPERINDEX': [ p[1], p[3] ], 'attr': atributosNil.copy() }

def p_expressionB3(p):
    'B : C SUBINDEX C'
    p[0] = { 'SUBINDEX': [ p[1], p[3] ], 'attr': atributosNil.copy() }

def p_expressionB4(p):
    'B : C SUPERINDEX C SUBINDEX C'
    p[0] = { 'SUPERSUBINDEX': [ p[1], p[3], p[5] ], 'attr': atributosNil.copy()
        }

def p_expressionB5(p):
    'B : C SUBINDEX C SUPERINDEX C'
    p[0] = { 'SUBSUPERINDEX': [ p[1], p[3], p[5] ], 'attr': atributosNil.copy()
        }

# C -> { E }
def p_expression_C1(p):
    'C : LBRACKET E RBRACKET'
    p[0] = p[2]

# C -> ( E )
def p_expression_C2(p):
    'C : LPAREN E RPAREN'
    p[0] = { '()': [ p[2] ], 'attr': atributosNil.copy() }

# C -> ID
# tengo que escribir el valor del token ID
```

```
def p_expression_C3(p):
    'C : ID'
    p[0] = { 'ID': p[1] , 'attr': atributosNil.copy() }



# Error rule for syntax errors
def p_error(p):
    print("Syntax error in input!")

# Cada key del diccionario tiene como valor una tupla < hijos, atributos >

def recorrer(t,tactual):    # primer recorrido top-down (rellenar tam) +++
    recorrido bottom-up (rellenar a, h1 y h2)

  t['attr']['h1'] = 0
  t['attr']['h2'] = 0

  if ('ID' in t.keys() ):
    t['attr']['tam'] = tactual
    t['attr']['a'] = t['attr']['tam'] * 0.6    # porque el ancho es 0.6 * tam

  if ( 'DIVIDE' in t.keys() ):
    t['attr']['tam'] = tactual * 2        # esto no se si esta bien. no depende
        de nom y denom ?
    elems = t.get('DIVIDE')
    nominador = recorrer(elems[0], tactual).copy()
    denominador = recorrer(elems[1], tactual).copy()

    nominador['attr']['a'] = elems[0]['attr']['a']
    denominador['attr']['a'] = elems[1]['attr']['a']

    ancho_divide = max(nominador['attr']['a'], denominador['attr']['a'])
    t['attr']['a'] = ancho_divide

    t['attr']['h1'] = denominador['attr']['tam']
    t['attr']['h2'] = nominador['attr']['tam']

    t['attr']['tam'] = nominador['attr']['tam'] + denominador['attr']['tam']

  if ('()' in t.keys() ):
    # t['attr']['tam'] = tactual    # este valor se lo pongo en la primera
        recorrida. en la 2da, lo modifico ( ? )
    elems = t.get('()')
    recorrer(elems[0],tactual)
    t['attr']['tam'] = elems[0]['attr']['tam']
    t['attr']['a'] = elems[0]['attr']['a'] + 0.6  # doy un margen a izq y
        derecha
    t['attr']['h1'] = elems[0]['attr']['h1']
    t['attr']['h2'] = elems[0]['attr']['h2']

  if ('CONCAT' in t.keys() ):
    t['attr']['tam'] = tactual
    ancho_concat = 0
    for elem in t.get('CONCAT'):
        h = recorrer(elem, tactual).copy()
        ancho_concat += h['attr']['a']
        t['attr']['tam'] = max(t['attr']['tam'], h['attr']['tam'])  # ACTUALIZO
            EL TAMANNO DE LA CONCATENACION
        t['attr']['h1'] = max(t['attr']['h1'], h['attr']['h1']) # ACTUALIZO EL
            TAMANNO DE LA CONCATENACION
        t['attr']['h2'] = max(t['attr']['h2'], h['attr']['h2'])  # ACTUALIZO EL
            TAMANNO DE LA CONCATENACION
```

```
    t['attr']['a'] = ancho_concat

  if ('SUPERINDEX' in t.keys() ):
    t['attr']['tam'] = 1.7 * tactual
    elems = t.get('SUPERINDEX')
    h1 = recorrer(elems[0], tactual).copy()
    h2 = recorrer(elems[1], tactual * 0.7).copy()
    t['attr']['tam'] = h1['attr']['tam'] + h2['attr']['tam'] # Actualizo el
        valor de tam por si alguno crecio.
    t['attr']['a'] = h1['attr']['a'] + h2['attr']['a']
    t['attr']['h1'] = h2['attr']['h1']
    t['attr']['h2'] = 0.6 * h1['attr']['tam'] + h2['attr']['h2']

  if ('SUBINDEX' in t.keys() ):
    t['attr']['tam'] = 1.7 * tactual
    elems = t.get('SUBINDEX')
    h1 = recorrer(elems[0], tactual).copy()
    h2 = recorrer(elems[1], tactual * 0.7).copy()
    t['attr']['tam'] = h1['attr']['tam'] + h2['attr']['tam'] # Actualizo el
        valor de tam por si alguno crecio.
    t['attr']['a'] = h1['attr']['a'] + h2['attr']['a']
    t['attr']['h1'] = 0.5 * h1['attr']['tam'] + h2['attr']['h1']
    t['attr']['h2'] = h2['attr']['h2']

  if ('SUPERSUBINDEX' in t.keys() ):
    t['attr']['tam'] = tactual + 0.7 * tactual + 0.7 * tactual
    elems = t.get('SUPERSUBINDEX')
    h1 = recorrer(elems[0], tactual).copy()
    h2 = recorrer(elems[1], tactual * 0.7).copy()
    h3 = recorrer(elems[2], tactual * 0.7).copy()
    t['attr']['tam'] = h1['attr']['tam'] + h2['attr']['tam']+ h3['attr']['tam']
        # Actualizo el valor de tam por si alguno crecio.
    t['attr']['a'] = h1['attr']['a'] + max(h2['attr']['a'], h3['attr']['a'])
    t['attr']['h1'] = 0.5 * h1['attr']['tam'] + h3['attr']['h1']
    t['attr']['h2'] = 0.6 * h1['attr']['tam'] + h2['attr']['h2']


  if ('SUBSUPERINDEX' in t.keys() ):
    t['attr']['tam'] = tactual + 0.7 * tactual + 0.7 * tactual
    elems = t.get('SUBSUPERINDEX')
    h1 = recorrer(elems[0], tactual).copy()
    h2 = recorrer(elems[1], tactual * 0.7).copy()
    h3 = recorrer(elems[2], tactual * 0.7).copy()
    t['attr']['tam'] = h1['attr']['tam'] + h2['attr']['tam']+ h3['attr']['tam']
        # Actualizo el valor de tam por si alguno crecio.
    t['attr']['a'] = h1['attr']['a'] + max(h2['attr']['a'], h3['attr']['a'])
    t['attr']['h1'] = 0.5 * h1['attr']['tam'] + h2['attr']['h1']
    t['attr']['h2'] = 0.6 * h1['attr']['tam'] + h3['attr']['h2']

  return t


def recorrer2(t, x, y): # segundo recorrido top-down, ahora para definir
    valores de x e y
                        # tambien modifico el tamanno de '()' para escalar lo
                            que hay entre '()'

    t['attr']['x'] = x
    t['attr']['y'] = y

    if ( 'DIVIDE' in t.keys() ):
        elems = t.get('DIVIDE')
```

```
        num_x = x
        den_x = x

        if (elems[0]['attr']['a'] < elems[1]['attr']['a']):
            num_x = t['attr']['x'] + 0.5 * (elems[1]['attr']['a'] - elems[0]['
                attr']['a'])
        else:
            den_x = t['attr']['x'] + 0.5 * (elems[0]['attr']['a'] - elems[1]['
                attr']['a'])

        num_y = t['attr']['y'] + elems[0]['attr']['h1'] + 0.05

        num = recorrer2(elems[0],num_x,num_y).copy()

        den_y = t['attr']['y'] - (elems[1]['attr']['tam'] - elems[1]['attr']['
            h1']) * 0.7
        den = recorrer2(elems[1],den_x,den_y).copy()


    if ( '()' in t.keys() ):

        elems = t.get('()')
        t['attr']['tam'] = elems[0]['attr']['tam']    # a '()' le pongo =
            tamanno que lo que hay adentro

        recorrer2(elems[0], x + 0.3 , y)   # faltaria desplazarse un poco por
            el '(' ?


    if ( 'CONCAT' in t.keys() ):
        elems = t.get('CONCAT')

        tam_0 = elems[0]['attr']['tam']
        tam_1 = elems[1]['attr']['tam']

        y_0 = y
        y_1 = y

        # FALTA NIVELAR EL Y CUANDO TENEMOS DIVISIONES
        # if( ('DIVIDE' in elems[0].keys()) and ('DIVIDE' not in elems[1].keys
            ()) ):
        #     y_1 = y - 0.3
        #     recorrer2(elems[0], x, y)
        #     recorrer2(elems[1], x + elems[0]['attr']['a'] , y_1)
        # elif( ('DIVIDE' in elems[1].keys()) and ('DIVIDE' not in elems[0].
            keys()) ):
        #     y_0 = y - 0.3
        #     recorrer2(elems[0], x, y_0)
        #     recorrer2(elems[1], x + elems[0]['attr']['a'] , y)
        # else:
        #     recorrer2(elems[0], x, y_0)
        #     recorrer2(elems[1], x + elems[0]['attr']['a'] , elems[0]['attr
            ']['y'])

        recorrer2(elems[0], x, y_0)
        recorrer2(elems[1], x + elems[0]['attr']['a'] , y_1)

        # t['attr']['y'] = min(elems[0]['attr']['y'], elems[1]['attr']['y'])

    if ( 'SUPERINDEX' in t.keys() ):
        elems = t.get('SUPERINDEX')
        h1 = recorrer2(elems[0], x, y).copy()
        recorrer2(elems[1], x + elems[0]['attr']['a'], y + t['attr']['h2'] -
```

```
            elems[1]['attr']['h2'])


    if ( 'SUBINDEX' in t.keys() ):
        elems = t.get('SUBINDEX')
        h1 = recorrer2(elems[0], x, y).copy()
        recorrer2(elems[1], x + elems[0]['attr']['a'], y - t['attr']['h1'] +
            elems[1]['attr']['h1'])

    if ( 'SUPERSUBINDEX' in t.keys() ):
        elems = t.get('SUPERSUBINDEX')
        h1 = recorrer2(elems[0], x, y).copy()
        recorrer2(elems[1], x + elems[0]['attr']['a'], y + t['attr']['h2'] -
            elems[1]['attr']['h2'])
        recorrer2(elems[2], x + elems[0]['attr']['a'], y - t['attr']['h1'] +
            elems[2]['attr']['h1'])

    if ( 'SUBSUPERINDEX' in t.keys() ):
        elems = t.get('SUBSUPERINDEX')
        h1 = recorrer2(elems[0], x, y).copy()
        recorrer2(elems[1], x + elems[0]['attr']['a'], y - t['attr']['h1'] +
            elems[1]['attr']['h1'])
        recorrer2(elems[2], x + elems[0]['attr']['a'], y + t['attr']['h2'] -
            elems[2]['attr']['h2'])


    return t



# === Parseo ====

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>'''


###
def recorrer3(t, out):

  if ( '()' in t.keys() ):
    elems = t.get('()')
    out.append( make_open_paren(str(0),str(0),
                                str(1),
                                str(elems[0]['attr']['x'] - 0.45),str(- elems
                                    [0]['attr']['y'] + elems[0]['attr']['h1'] *
                                    0.6),
                                str(1),str(elems[0]['attr']['tam']*1) ) )
    recorrer3(elems[0], out)
```

```
    out.append( make_close_paren(str(0),str(0),
                                 str(1),
                                 str(elems[0]['attr']['x'] + elems[0]['attr']['a
                                     '] - 0.15),str(- elems[0]['attr']['y'] +
                                     elems[0]['attr']['h1'] * 0.6),
                                 str(1),str(elems[0]['attr']['tam']*1) ) )

  if ( 'DIVIDE' in t.keys() ):
    elems = t.get('DIVIDE')
    recorrer3(elems[0], out)

    out.append( make_line(str(t['attr']['x']), str(-t['attr']['y']),
                          str(t['attr']['x'] + t['attr']['a']), # recien aca
                              balanceo el largo de la linea de div.
                          str(-t['attr']['y']), str(0.05)) )  # harcodeo el
                              ancho de la linea de division

    recorrer3(elems[1], out)

  if ( 'ID' in t.keys() ):
    out.append ( make_text(str(t['attr']['x']), str(-t['attr']['y']),   # me
        muevo al reves sobre el eje y
                           str(t['attr']['tam']), str(t['ID']) ) )

  if ( 'CONCAT' in t.keys() ):
    elems = t.get('CONCAT')
    recorrer3(elems[0], out)
    recorrer3(elems[1], out)

  if ( 'SUPERINDEX' in t.keys() ):
    elems = t.get('SUPERINDEX')
    recorrer3(elems[0], out)
    recorrer3(elems[1], out)

  if ( 'SUBINDEX' in t.keys() ):
    elems = t.get('SUBINDEX')
    recorrer3(elems[0], out)
    recorrer3(elems[1], out)

  if ( 'SUPERSUBINDEX' in t.keys() ):
    elems = t.get('SUPERSUBINDEX')
    recorrer3(elems[0], out)
    recorrer3(elems[1], out)
    recorrer3(elems[2], out)

  if ( 'SUBSUPERINDEX' in t.keys() ):
    elems = t.get('SUBSUPERINDEX')
    recorrer3(elems[0], out)
    recorrer3(elems[1], out)
    recorrer3(elems[2], out)
```