Teoría de Lenguajes

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

Trabajo Práctico

Integrante	LU	Correo electrónico
Claverino, Daniel	273/10	dclave@gmail.com
Conde, Fernando	423/09	ferconde87@hotmail.com
Forte, Martín	363/10	martinforte@yahoo.com.ar

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introduccion

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. Gramática

Las producciones de la gramática que se nos fue presentada son de la forma:

E	→	EE
	|	E∧E
	|	E_E
	|	E∧E_E
	|	E_E∧E
	|	E/E
	|	(E)
	|	{E}
	|	l

Como dicha gramática tiene problemas de ambigüedad y hay ciertas restricciones del lenguaje que no están contempladas, decidimos modificarla. A continuación presentamos nuestra gramática y la explicación de por qué representa el mismo lenguaje:

El símbolo distiguido ahora es S, el cual deriva en E, y nos permite saber cuándo se reduce a la raíz. Uno de los problemas que teníamos con la ambigüedad es que E podía ser cualquier tipo de expresión, con lo cual decidimos darle una cierta jerarquía a cada expresión (aprovechando que la gramática lo solicitaba) y de este modo deshacernos de la ambigüedad.

S	→	E
E	→	E / CONCAT | CONCAT
CONCAT	→	CONCAT ELEMENTS | ELEMENTS
ELEMENTS	→	INDEXES | NOINDEX

E puede convertirse en divisiones o bien pasar a ser una $CONCAT$enación recursiva de $ELEMENTS$ donde cada elemento de la concatenación puede ser una expresión que contiene $INDEXES$ o bien $NOINDEXES$.

$INDEXES$, como su nombre lo indica, contiene todas las expresiones que tienen índices. No pueden estar directamente formadas por expresiones $INDEXES$ ya que los indices no son asociativos (por enunciado) y porque la concatenación de subíndices con superíndices generaría ambigüedad, por ejemplo con la expresión original E → E_E∧E.

INDEXES	→	SUPER | SUB | SUPSUB | SUBSUP
SUPER	→	NOINDEX∧NOINDEX
SUB	→	NOINDEX_NOINDEX
SUPSUB	→	NOINDEX∧NOINDEX_NOINDEX
SUBSUP	→	NOINDEX_NOINDEX∧NOINDEX

$NOINDEX$ son todas aquellas expresiones que no son formadas por índices: ID, PAR y $GROUP$.

NOINDEX	→	PAR | GROUP | ID
PAR	→	(E)
GROUP	→	{E}
ID	→	l

3. Manual de uso

En esta sección veremos los requerimientos para correr el programa y cómo utilizarlo. Los requerimientos son:

- Python 2.7
- ply-3.8
- Algún visor de SVG para poder visualizar el output

El programa se puede correr de distintas formas:

- Completamente interactivo:

  Tanto el input como el output es dentro del programa. Para ello basta con ejecutarlo del siguiente modo: `python main.py`

- Input por línea de comando:

  La fórmula se introducirá por linea de comandos pero el resultado se verá por pantalla.

  Para esto ejecutar: `python main.py --formula "FORMULA"` donde FORMULA es la fórmula que se desee convertir. Las comillas **no** son opcionales ya que el módulo de Python con el que parametrizamos la entrada parece no comprender el símbolo de los superindices.

- Output por file:

  La fórmula se introduce dentro del programa pero el output se especifica por linea de comandos: PYTHON MAIN.PY --OUTPUT FILENAME

  Ejemplo: `python main.py --output salida.svg`

- Línea de comandos:

  El input y el output se especifica en el comando para esto: `python main.py --formula "FORMULA"--output FILENAME`

4. Código

En **main.py** usando *ply* tenemos definidos el *lexer* y el *parser*, una funcion encargada de los tests y el programa propiamente dicho. El Lexer no merece mucha atención, así que pasaremos directamente al parser:

Como ply utiliza LALR y éste sólo permite atributos sintetizados, decidimos tener un único atributo que va formando un árbol a medida que se hacen las reducciones. Dicho árbol no refleja la sintaxis, sino que es más bien una representación de las posiciones y tamaños de los caracteres de la fórmula, junto con su dependencia.

En **Node.py** está el código de los Nodos de este árbol.

Tomando como ejemplo la producción $CONCAT_1 \rightarrow CONCAT_2\ ELEMENTS$, donde en ply se separan los terminales y no-terminales como $p[0]$ ($CONCAT_1$), $p[1]$ ($CONCAT_2$) y $p[2]$ ($ELEMENTS$), lo que hacemos es $p[0] = ConcatNode(p[1], p[2])$. Es decir, representamos la concatenación de los dos no-terminales ya procesados como árboles en $p[1]$ y $p[2]$.

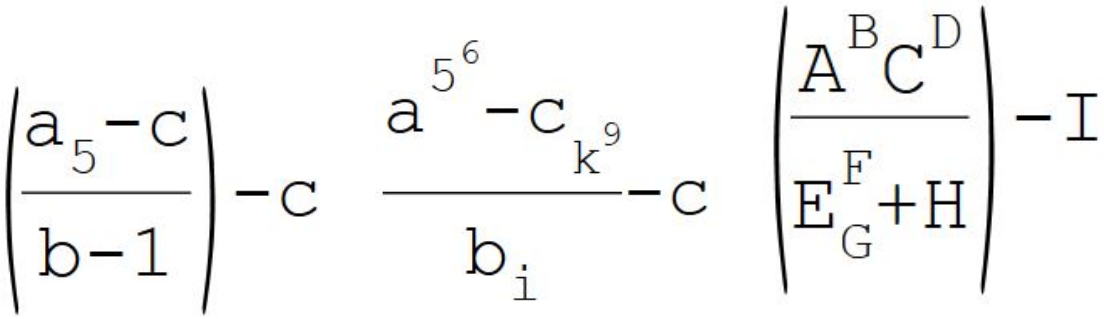
Por otro lado, en $GROUP \rightarrow \{E\}$, tenemos $p[0]$ ($GROUP$), $p[1]$ ({), $p[2]$ (E) y $p[3]$ (}), y en este caso pasamos directamente el árbol procesado en $p[2]$. Es decir, $p[0] = p[2]$, ya que no hay ningún cambio respecto a la posicion o tamaño.

Estos dos ejemplos muestran que el árbol que generamos tiende a ser sintáctico pero no lo es necesariamente. De hecho, definimos un `LineNode` que representa la línea de división, y que se instancia en la implementación de `DivideNode`.

Finalmente, en la producción $S \rightarrow E$ simplemente completamos el árbol con la raíz y llamamos al método $toSvg()$ de ésta. $p[0] = MainNode(p[1]).toSvg()$, donde $MainNode$ representa la raíz del árbol y $toSvg()$ va generando y concatenando los $toSvg$ de los subnodos para obtener el svg final de la fórmula.

A continuación incluímos unos ejemplos de resultados obtenidos y luego presentamos el código de **main.py** y **Node.py**:

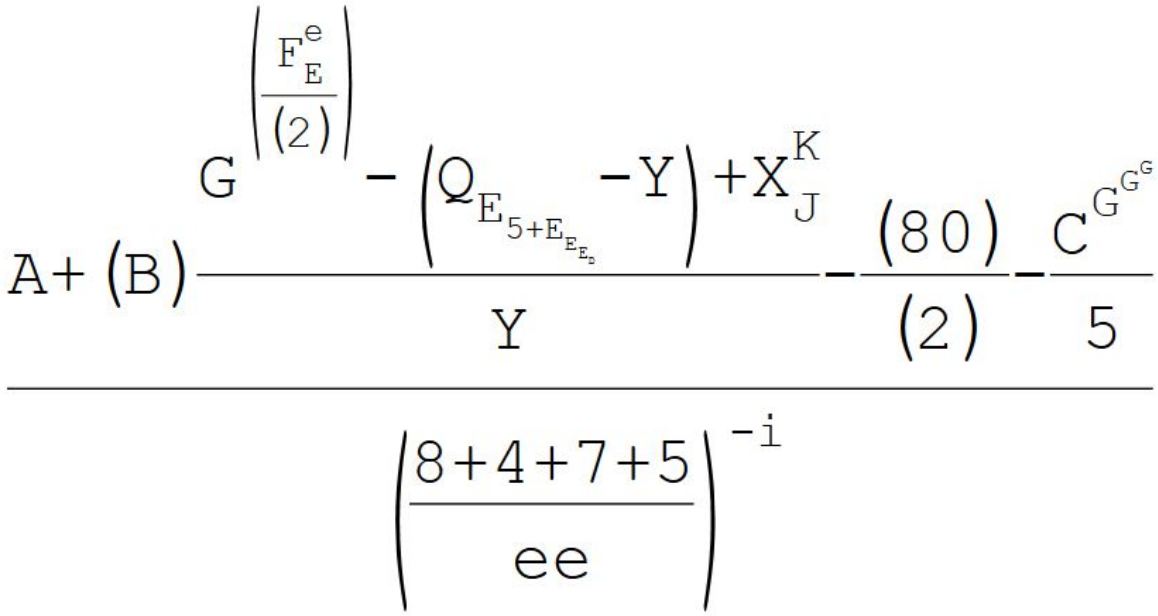
4.1. Ejemplos



(a) $(a\_5 - c/b - 1) - c$

(b) $\{a \wedge \{5 \wedge 6\} - c\_\{k \wedge 9\}/b\_i\} - c$

(c) $(A \wedge BC \wedge D/E \wedge F\_G + H) - I$



(d) $A + (B)\{G \wedge \{(F \wedge e\_E/(2))\} - (Q\_\{E\_\{\{5\} + E\_\{E\_\{E\_D\}\}\}\} - Y) + X \wedge K\_J/Y\} - \{(80)/(2)\} - \{C \wedge \{G \wedge \{G \wedge \{G\}\}\}/5\}/(\{8 + 4 + 7\} + 5/ee) \wedge \{-i\}$

4.2. main.py

```
import argparse
import sys

from Node import MainNode
from Node import DivideNode
from Node import ConcatNode
from Node import SuperIndexNode
from Node import SubIndexNode
from Node import SuperSubIndexNode
from Node import CharacterNode
from Node import ParenthesisNode


#==========================================
#
# LEXER
#
#==========================================

tokens = (
    'SUB',
    'SUPER',
    'DIVIDE',
    'LPAREN',
    'RPAREN',
    'LLLAVE',
    'RLLAVE',
    'CARACTERES'
)
# Tokens

t_SUB    = r'_'
t_SUPER    = r'\^'
t_DIVIDE   = r'/'
t_LPAREN   = r'\('
t_RPAREN   = r'\)'
t_LLLAVE   = r'\{'
t_RLLAVE   = r'\}'
t_CARACTERES = r'[^_\^/\(\)\{\}]'



# Ignored characters
t_ignore = " \t"

def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")

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

# Build the lexer
import ply.lex as lex
lex.lex()
```

```
#============================================
#
# PARSER
#
#============================================

# S -> E
def p_statement_expression(p):
    'statement : expression'
    p[0] = MainNode(p[1]).toSvg()

# E -> E / CONCAT
def p_expression_divide(p):
    'expression : expression DIVIDE concat'
    p[0] = DivideNode(p[1], p[3])

# E -> CONCAT
def p_expression_concat(p):
    'expression : concat'
    p[0] = p[1]

# CONCAT -> CONCAT ELEMENTS
def p_concat_concat(p):
    'concat : concat elements'
    p[0] = ConcatNode(p[1], p[2])

# CONCAT -> ELEMENTS
def p_concat_elements(p):
    'concat : elements'
    p[0] = p[1]

# ELEMENTS -> INDEXES
def p_elements_indexes(p):
    'elements : indexes'
    p[0] = p[1]

# ELEMENTS -> NOINDEX
def p_elements_noindex(p):
    'elements : noindex'
    p[0] = p[1]

# SUPER -> NOINDEX^NOINDEX
def p_super(p):
    'super : noindex SUPER noindex'
    p[0] = SuperIndexNode(p[1], p[3])

# SUB -> NOINDEX_NOINDEX
def p_sub(p):
    'sub : noindex SUB noindex'
    p[0] = SubIndexNode(p[1], p[3])

# SUPSUB -> NOINDEX^NOINDEX_NOINDEX
```

```
def p_superindex_subindex(p):
    'supsub : noindex SUPER noindex SUB noindex'
    p[0] = SuperSubIndexNode(p[1], p[3], p[5])

# SUBSUP -> NOINDEX_NOINDEX^NOINDEX
def p_subindex_superindex(p):
    'subsup : noindex SUB noindex SUPER noindex'
    p[0] = SuperSubIndexNode(p[1], p[5], p[3])

# INDEXES -> SUPER | SUB | SUPSUB | SUBSUP
def p_indexes(p):
    '''indexes : super
               | sub
               | supsub
               | subsup'''
    p[0] = p[1]

#NOINDEX -> PAR | GROUP | ID
def p_noindex(p):
    '''noindex : parenthesis
               | group
               | id'''
    p[0] = p[1]

# ID -> l
def p_id(p):
    'id : CARACTERES'
    p[0] = CharacterNode(p[1])

# PAR -> (E)
def p_parenthesis(p):
    'parenthesis : LPAREN expression RPAREN'
    p[0] = ParenthesisNode(p[2])

# GROUP -> {E}
def p_group(p):
    'group : LLLAVE expression RLLAVE'
    p[0] = p[2]

def p_error(p):
    if(p):
        if(s): # Si esta seteada la variable global la uso
               #   para expresar mejor el error
            if(len(s) > p.lineno+1):
                error = "Error en el caracter '%s'. Contexto: '%s' '%s' '%s'" \
                    %(p.value, s[0:p.lineno], s[p.lineno], s[p.lineno+1:])
            else:
                error = "Error en el caracter '%s'. Contexto: '%s'" \
                    %(p.value, s[0:p.lineno])
        else:
            error = "Error en el caracter '%s' en la posicion %d." \
                %(p.value, p.lineno)
    else: # en algunos casos p no viene definido y no hay mucha mas
          #  informacion para mostrar
        error = "Error de sintaxis."
    raise SyntaxError(error)

import ply.yacc as yacc
```

```
yacc.yacc()



#==========================================
#
# TESTS
#
#==========================================

def test():
    # Casos que tiene que tiene que reconocer
    assert test_accept('(a_5-c/b-1)-c')
    assert test_accept('{a^5-c/b}-c')
    assert test_accept('{a^{5^6}-c_{{k^9}}/b_i}-c')
    assert test_accept('(10+5/2)')
    assert test_accept('1_2^{3_4^{5_6^7}}')
    assert test_accept('(A^BC^D/E^F_G+H)-I')
    assert test_accept('A+(B){G^{(F^e_E/(2))}-\
        (Q_{E_{{5}+E_{E_{E_D}}}}-Y)+X^K_J/Y}-\
        {(80)/(2)}-{C^{G^{G^{G}}}/5}/({8+4+7}+5/ee)^{-i}')

    assert test_not_accept('1^2^3')
    assert test_not_accept('1_2_3')
    assert test_not_accept('_')
    assert test_not_accept('_1')
    assert test_not_accept('^')
    assert test_not_accept('^1')
    assert test_not_accept('1_')
    assert test_not_accept('1/')
    assert test_not_accept('1/^')
    assert test_not_accept('{1+2')
    assert test_not_accept('1+2}')
    assert test_not_accept('(1+2')
    assert test_not_accept('3(1+2')
    assert test_not_accept('1+2)')
    assert test_not_accept('()')
    assert test_not_accept('())')
    assert test_not_accept('{{1}')


def test_not_accept(test):
    success = True
    try:
        s = yacc.parse(test)
        success = False # Si no falla paso por aca
    except SyntaxError:
        pass
    return success


# Si no pincha es porque se parseo correctamente
def test_accept(test):
        try:
                s = yacc.parse(test)
        except SyntaxError:
                return False
        return True
```

```
# Setea la variable global para poder mostrar
#   mas informacion del error de parseo
s = None
test()



#==========================================
#
# PROGRAMA
#
#==========================================

# Parseo de la entrada
parser = argparse.ArgumentParser(description='Conversor de formulas a SVG.')
parser.add_argument('--formula', type=str, \
    help='Formula en formato pseudo latex')
parser.add_argument('--output', type=str, \
    help='nombre del archivo donde se guardara el resultado')
args = parser.parse_args()

if(args.formula):
    s = args.formula
else:
    try:
        s = raw_input('Formula :> ')
    except EOFError:
        pass

try:
    svg = yacc.parse(s)
except SyntaxError as e:
    # Si hay un error de parseo lo muestro por el standard error
    print >> sys.stderr, e
    raise SystemExit

if(args.output):
    with open(args.output, 'w') as f:
        f.write(svg)
else:
    print(svg)
```

4.3. Node.py

```
FONT_SIZE             = 48                #Tamanio por defecto
CHAR_WIDTH            = .6 * FONT_SIZE    #Ancho de un caracter
CHAR_HEIGHT           = 1.0 * FONT_SIZE   #Alto de un caracter
CHAR_UPPER_HEIGHT     = .8 * FONT_SIZE    #Alto sobre baseline
CHAR_LOWER_HEIGHT     = .2 * FONT_SIZE    #Alto por debajo del baseline
PARENTHESIS_SPACING   = .5 * FONT_SIZE    #Espaciado del parentesis
LINE_SPACING          = .1 * FONT_SIZE    #Espaciado de la linea
SUPER_SPACING         = -0.5 * CHAR_UPPER_HEIGHT  #Espaciado (eje Y) superindice
SUB_SPACING           = -0.5 * CHAR_UPPER_HEIGHT  #Espaciado (eje Y) subindice

class Node(object):
    def __init__(self, wid, hlw, hup):
        self.x = 0                  #Posicion X absoluta
        self.y = 0                  #Posicion Y absoluta
        self.wid = wid              #Ancho
        self.hlw = hlw              #Altura por encima del baseline
        self.hup = hup              #Altura por debajo del baseline
        self.scale = FONT_SIZE      #Tamanio de la fuente
        self.subNodes = []          #Subnodos

    def getPosition(self):
        return (self.x, self.y)     #Posicion absoluta

    def getHeights(self):
        return (self.hlw, self.hup)

    def getLowerHeight(self):
        return self.hlw

    def getUpperHeight(self):
        return self.hup

    def getHeight(self):
        return self.hup + self.hlw

    def getWidth(self):
        return self.wid

    def getScale(self):
        return self.scale

    #Modifica la posicion de este nodo y sus subnodos
    def setPosition(self, position):
        nx, ny = position
        dx = nx - self.x
        dy = ny - self.y
        self.movePosition(dx, dy)

    #Escala este nodo y sus subnodos
    def scaleBy(self, scale):
        self.scale *= scale
        self.x *= scale
        self.y *= scale
        self.wid *= scale
        self.hup *= scale
        self.hlw *= scale
```

```
        for node in self.subNodes:
            node.scaleBy(scale)

    #Desplaza este nodo y sus subnodos
    def movePosition(self, x, y):
        self.x += x
        self.y += y
        for node in self.subNodes:
            node.movePosition(x, y)

    def addNode(self, node):
        self.subNodes.append(node)

    def toSvg(self):
        return "".join([node.toSvg() for node in self.subNodes])

class MainNode(Node):
    def __init__(self, node):
        wid = node.getWidth()
        hlow, hupp = node.getHeights()
        super(MainNode, self).__init__(wid, hlow, hupp)
        self.addNode(node)

    def toSvg(self):
        return '''<?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(10, %.2f) " font-family="Courier">\n''' \
                %(10+ self.hup - self.y) + \
            super(MainNode, self).toSvg() + \
            '''</g></svg>'''

class CharacterNode(Node):
    def __init__(self, char):
        super(CharacterNode, self).__init__(CHAR_WIDTH, \
                CHAR_LOWER_HEIGHT, CHAR_UPPER_HEIGHT)
        self.char = char

    def toSvg(self):
        x, y = self.getPosition()
        return "<text x='%.2f' y='%.2f' font-size='%.2f'>%s</text>\n" \
            %(x, y, self.scale, self.char)

class ConcatNode(Node):
    def __init__(self, nodeA, nodeB):
        wid = nodeA.getWidth() + nodeB.getWidth()
        hlowA, huppA = nodeA.getHeights()
        hlowB, huppB = nodeB.getHeights()

        super(ConcatNode, self).__init__(wid, \
                max(hlowA, hlowB), max(huppA, huppB))

        #Mueve el nodo
        self.setPosition(nodeA.getPosition())

        #Agrega el 1er nodo
        self.addNode(nodeA)
```

```
        #Agrega el 2do nodo
        self.addNode(nodeB)

        #Mueve el nodo y B a la pos. de A
        nodeB.setPosition(nodeA.getPosition())

        #Desplaza B.x por A.width()
        nodeB.movePosition(nodeA.getWidth(), 0)

class ParenthesisNode(Node):
    def __init__(self, subNode):
        wid = subNode.getWidth() + PARENTHESIS_SPACING * 2
        hlow, hupp = subNode.getHeights()

        super(ParenthesisNode, self).__init__(wid, hlow, hupp)
        self.setPosition(subNode.getPosition())
        subNode.movePosition(PARENTHESIS_SPACING, 0)
        self.addNode(subNode)

    def toSvg(self):
        x, y = self.getPosition()
        scale = self.getScale()

        #Magic number that works
        height = self.getHeight() / .74

        #Me fijo la posicion del subnodo
        snx, _ = self.subNodes[0].getPosition()

        #Y le resto la posicion del parentesis para saber el spacing
        #  a la izquierda.
        #Sacandole al width total este spacing, queda la distancia al
        #  parentesis derecho
        subNodeWidth = self.getWidth() - (snx - x)

        y += self.getLowerHeight()
        y -= (0.12 * FONT_SIZE) * height / scale

        parStr = "<text x='0' y='0' font-size='%.2f' \
            transform='translate(%.2f, %.2f) \
            scale(1, %.2f)'>%s</text>\n"
        lPar = parStr %(scale, x, y, height / scale, '(')
        rPar = parStr %(scale, x + subNodeWidth, y, height / scale, ')')

        return lPar + super(ParenthesisNode, self).toSvg() + rPar

class SubIndexNode(Node):
    def __init__(self, rootNode, indexNode):
        indexNode.scaleBy(0.7)

        wid = rootNode.getWidth() + indexNode.getWidth()
        hlow, hupp = rootNode.getHeights()

        hlow += indexNode.getHeight() + SUB_SPACING
        super(SubIndexNode, self).__init__(wid, hlow, hupp)
```

```
        #Mueve el nodo Index a la derecha y un poco mas abajo que Root
        indexNode.movePosition(rootNode.getWidth(), indexNode.getUpperHeight() \
            + rootNode.getLowerHeight() + SUB_SPACING)

        #Agrega el nodo Index
        self.addNode(indexNode)

        #Setea este nodo e Index en la posicion de Root
        self.setPosition(rootNode.getPosition())

        self.addNode(rootNode)

class SuperIndexNode(Node):
    def __init__(self, rootNode, indexNode):
        indexNode.scaleBy(0.7)

        wid = rootNode.getWidth() + indexNode.getWidth()
        hlow, hupp = rootNode.getHeights()

        hupp += indexNode.getHeight() + SUPER_SPACING
        super(SuperIndexNode, self).__init__(wid, hlow, hupp)

        #Mueve el nodo Index a la derecha y un poco mas arriba que Root
        indexNode.movePosition(rootNode.getWidth(), -indexNode.getLowerHeight() \
            - rootNode.getUpperHeight() - SUPER_SPACING)

        #Agrega el nodo Index
        self.addNode(indexNode)

        #Setea este nodo e Index en la posicion de Root
        self.setPosition(rootNode.getPosition())

        self.addNode(rootNode)

class SuperSubIndexNode(Node):
    def __init__(self, rootNode, superNode, subNode):
        superNode.scaleBy(0.7)
        subNode.scaleBy(0.7)

        wid = rootNode.getWidth() + max(superNode.getWidth(), subNode.getWidth())
        hlow, hupp = rootNode.getHeights()
        hlow += subNode.getHeight() + SUB_SPACING
        hupp += superNode.getHeight() + SUPER_SPACING
        super(SuperSubIndexNode, self).__init__(wid, hlow, hupp)

        #Mueve el Sub a la derecha y abajo del Root
        subNode.movePosition(rootNode.getWidth(), subNode.getUpperHeight() \
            + rootNode.getLowerHeight() + SUB_SPACING)

        #Mueve el Super a la derecha y arriba del Root
        superNode.movePosition(rootNode.getWidth(), -superNode.getLowerHeight() \
            - rootNode.getUpperHeight() - SUPER_SPACING)


        #Agrega el nodo Sub
        self.addNode(subNode)

        #Agrega el nodo Super
```

```
        self.addNode(superNode)

        #Setea este nodo, Sub y Super en la posicion de Root
        self.setPosition(rootNode.getPosition())


        self.addNode(rootNode)

class DivideNode(Node):
    def __init__(self, upperNode, lowerNode):
        uwid = upperNode.getWidth()
        lwid = lowerNode.getWidth()
        if uwid > lwid:
            wid = uwid
        else:
            wid = lwid
        hlow = lowerNode.getHeight() + CHAR_HEIGHT * 0.28
        hupp = upperNode.getHeight() + CHAR_HEIGHT * 0.4

        #Instancio un nodo para la linea de division
        lineNode = LineNode(wid)


        super(DivideNode, self).__init__(wid, hlow, hupp)

        #Mueve el Nodo a la posicion de Upper
        self.setPosition(upperNode.getPosition())
        self.addNode(upperNode)
        self.addNode(lineNode)
        self.addNode(lowerNode)

        #Mueve la linea al Baseline
        lineNode.setPosition(upperNode.getPosition())

        #Mueve la linea un poco arriba para que quede alineada con los '-''
        lineNode.movePosition(0, -.28*CHAR_HEIGHT)

        #Mueve Lower al Baseline
        lowerNode.setPosition(upperNode.getPosition())

        #Mueve Lower por debajo de la linea
        lowerNode.movePosition(0, lowerNode.getUpperHeight())

        #Mueve Upper por encima de la linea
        upperNode.movePosition(0, -upperNode.getLowerHeight() -.4*CHAR_HEIGHT)


        #Centra el numerador o denominador
        if uwid > lwid:
            lowerNode.movePosition((uwid - lwid) / 2, 0)
        else:
            upperNode.movePosition((lwid - uwid) / 2, 0)


class LineNode(Node):
    def __init__(self, wid):
        super(LineNode, self).__init__(wid, LINE_SPACING, LINE_SPACING)
```

```
def toSvg(self):
    x, y = self.getPosition()
    wid = self.getWidth()
    lineStr = "<line x1='%.2f' y1='%.2f' x2='%.2f' y2='%.2f' \
        stroke-width='%.3f' stroke='black' />\n"
    stroke = 0.03 * self.scale
    return (lineStr %(x, y, x + wid, y, stroke)) \
        + super(LineNode, self).toSvg()
```

5. Conclusiones

Una de las cosas que notamos es que hay gramáticas que son ambiguas y eso las hace fćilmente comprendibles por humanos pero las computadoras «prefieren» gramáticas que no lo sean. Aprendimos que con un poco de trabajo podemos, no solo desambiguar una gramtica, sino también modificarla para que algunas restricciones que estaban dadas semánticamente queden abarcadas por la gramática como lo fueron la no asociatividad de los exponentes y la precedencia de algunos operadores.

Pudimos ver de una forma práctica como el parser interactúa con el lexer y semántica para formar cosas muy poderosas de una manera muy simple, ya que con pequeños ladrillitos como lo son las producciones se generan de forma recursiva cosas inimaginables. Después de este trabajo práctico, ver cosas como las que hace LaTeXnos dejaron de parecer «magia negra» y pudimos entender que el concepto por el cual se rigen es el mismo.