Trabajo Práctico

Teoría de Lenguajes

Segundo Cuatrimestre de 2017

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Integrante	LU	Correo electrónico
Bokser Brian	155/15	brian.bokser@gmail.com
Franco Lancioni	234/15	gianflancioni@gmail.com
Jonathan Scherman	152/15	jonischerman@gmail.com

Índice

1. Introducción

En este trabajo práctico desarrollamos un compositor de fórmulas matemáticas. El mismo toma como entrada la descripción de una fórmula en una versión simplificada del lenguaje utilizado por $LATEX$ y produce como salida un archivo SVG (Scalable Vector Graphics). Esta versión simplificada nos permite usar utilizar la división (o fracciones) y utilizar subíndices o superíndices.

Usamos la librería PLY [1] de Python que nos permite hacer tanto el análisis léxico, que consiste en identificar subexpresiones de una cadena texto con símbolos terminales de nuestra gramática, como sintáctico, que consiste en trabajar sobre la estructura relativa a dicha gramática de la cadena input. Modificamos la gramática dada por enunciado, agregando además precedencias para que sea LALR.

Hacia el final, hicimos pruebas de los resultados generados por nuestro programa frente a fórmulas variadas, con muy buenos resultados.

2. Desarrollo

2.1. Lexer

Utilizamos los siguientes tokens con sus respectivas reglas

- **DIVIDE**: El símbolo '/' para la división
- Los literales: "_ ^ () {}"
- **CHR**: Cualquier símbolo exceptuando los anteriores.
  Esto incluye:
  - cualquier caracter a-z y A-Z
  - +
  - *

2.2. Gramática

$\mathcal{G} = \langle \{E, UNARYEXP, SUPEREXP, SUBEXP\}, \{CHR, DIVIDE, \text{'\_'}, \text{'\^{}'}, \text{'('}, \text{')'}, \text{'\{'}, \text{'\}'} \}, \mathcal{P}, E \rangle$

$$
\begin{array}{rrcl}
\mathcal{P}: & E & \rightarrow & UNARYEXP \\
 & & | & E\ E \\
 & & | & E\ \textbf{DIVIDE}\ E \\
 & & | & UNARYEXP\ \hat{}\ UNARYEXP\ SUBEXP \\
 & & | & UNARYEXP\ \_\ UNARYEXP\ SUPEREXP \\
 & & | & E\ \texttt{DIVIDE}\ E \\
 UNARYEXP & & \rightarrow & \textbf{CHR} \\
 & & | & (\ E\ ) \\
 & & | & \{\ E\ \} \\
 SUPEREXP & & \rightarrow & \hat{}\ UNARYEXP \\
 & & | & \lambda \\
 SUBEXP & & \rightarrow & \_\ UNARYEXP \\
 & & | & \lambda
\end{array}
$$

Figura 1: Producciones de la gramática

Siendo la gramática original la siguiente:

$$
\begin{array}{rcll}
E & \rightarrow & E\ E & \text{(concatenación)} \\
 & | & E\ \hat{}\ E & \text{(superíndice)} \\
 & | & E\ \_\ E & \text{(subíndice)} \\
 & | & E\ \hat{}\ E\ \_\ E & \text{(super y subíndice)} \\
 & | & E\ \_\ E\ \hat{}\ E & \text{(sub y superíndice)} \\
 & | & E\ /\ E & \text{(división)} \\
 & | & (\ E\ ) & \text{(encerrar entre paréntesis)} \\
 & | & \{\ E\ \} & \text{(agrupar)} \\
 & | & l & \text{(cualquier caracter salvo \^{}, \_, /, \{, \}, (y))}
\end{array}
$$

Figura 2: Gramática original a parsear según enunciado.

Como *PLY*, la herramienta que usamos tanto para el análisis lexicográfico (lexer) como para parsear, usa técnicas de tablas LALR, tuvimos que hacer algunas modificaciones sobre la gramática original para poder generar una tabla de dichas características.

Una de ellas fue juntar las producciones *"superíndice"* y *"super y subíndice"* en una única usando un no-terminal nuevo $SUBEXP$ que pudiera ser anulable o generar el subíndice e idem para las producciones *"subíndice"* y *"sub y superíndice"*. De lo contrario una cadena con super y subíndices como $A\hat{}B\_C$ podría ser generada produciendo primero un superíndice y luego un subíndice usando dos producciones tanto como usando una única producción.

Otra manera de sortear este problema hubiera sido eliminando las producciones que combinan super y subíndices y usando ambas producciones de manera consecutiva, pero preferimos mantener las producciones ternarias sobre las binarias para facilitar el recorrido y "decorado" de la estructura sintáctica (sino se nos complicaría distinguir estructuras de cadenas como $A\_\{B\hat{}C\}$ ó $\{A\_B\}\hat{}C$ de las de $A\_B\hat{}C$).

El otro cambio importante fue agregar otro no-terminal $UNARYEXP$ que genera producciones 'unarias' (paréntesis o llaves sobre una E, o un CHR). La idea viene de la necesidad de desambiguar expresiones como $E\hat{}E\_E$ que podrían ser generadas como $E \Rightarrow E\hat{}E\ SUBEXP \Rightarrow E\hat{}E\_E$ o como $E \Rightarrow E\hat{}E\ SUBEXP \Rightarrow E\hat{}E \Rightarrow E\ \hat{}E\_\ E\ SUPEREXP \Rightarrow E\hat{}E\_\ E$.

Como según el enunciado ni los '^' ni los '_' son asociativos y además $E\_E\hat{}E$ y $E\hat{}E\_E$ son equivalentes, viendo expresiones de la pinta $E_1\_E_2\hat{}E_3$ se puede ver que ninguno de los tres no-terminales pueden producir nunca otro subíndice o superíndice en el mismo 'scope' de paréntesis o llaves. Esto es porque siempre podríamos invertir el orden usando la equivalencia mencionada anteriormente de modo que se asocien dos de estos símbolos.
Tampoco pueden producir concatenaciones o divisiones porque '^' y '_' tienen mayor precedencia que estos, por ejemplo la cadena $A\hat{}BC$ (escribiendo los terminales $CHAR$ como sus valores para mayor claridad) no tiene la estructura de $A\hat{}\{BC\}$ sino de $A\hat{}B$ concatenado a C [1] y en el caso de $A/B\hat{}C\_D$ primero se resuelve la indexación de B y luego la división por lo tanto la estructura se corresponde a $A/\{B\hat{}C\_D\}$ [2].

Con todos estos cambios aún seguimos teniendo problemas de tipo *Shift/Reduce* y *Reduce/Reduce* en nuestras tablas, que resolvimos declarando las siguientes precedencias:

Tabla de precedencia (en orden creciente)

- **DIV** asociativa a izquierda
- '{', '(' asociativas a izquierda

[1] idem para subindexación
[2] idem si fuera concatenación

- **CHR** asociativa a izquierda
- CONCAT, asociativa a izquierda, pseudosímbolo para la concatenación
- '^' no asociativa
- '_' no asociativa

Las reglas de la concatenación, división e indexación siguen la descripción del enunciado mientras que las que se corresponden a $Primeros(E)$ sirven para resolver en favor de *Shifts* cuando se llega al final de la expresión de una división y se está por ver una concatenación (es decir, **DIV** tiene menos precedencia que los 'primeros' de E y que las otras operaciones) y que se tome *Reduce* cada vez que se vió una concatenación si se está por ver una expresión concatenada o de división [3].

2.3. AST

Como *Yacc* solo nos permite sintetizar atributos únicamente en una "pasada" sobre el árbol de parsing y como veremos en la sección de atributos requerimos de varias pasadas para setear los atributos deseados, decidimos sintetizar como atributo de la gramática justamente su *AST* para luego poder recorrerlo múltiples veces decorándolo.

La idea es sencilla: por cada producción de la gramática sintetizamos un nodo que simboliza una operación sobre sus subtérminos. Por ejemplo, el input `A/B+C^D_E` se corresponde con la estructura:
`DivExpr(Chr(A),Concat(Concat(Chr(B),Chr(+)),SuperSub(Chr(C), Chr(D), SubSuffix(Chr(E)))))`

Cabe mencionar que no asignamos nodos a producciones de tipo 'E → UNARYEXP' dado que no aportan nada en cuanto a términos sintácticos. Lo mismo para las llaves que solamente sirven para indicar precedencias a la hora de parsear las cadenas.

2.4. Atributos y SVG

Para poder generar un SVG partiendo del AST ya sintetizado, necesitamos decorarlo con atributos:

- **e**: el interlineado o escala *(heredado)*
- **a**: ancho de una expresión *(sintetizado)*
- **h1**: corrimiento "para arriba" desde la base de una expresión *(sintetizado)*
- **h2**: corrimiento "para abajo" desde la base de una expresión *(sintetizado)*
- **h**: cuánto abarca de largo la expresión en total *(sintetizado)*
- **x**: ubicación en el eje x de una expresión *(heredado)*

[3] Esto en teoría alcanzaría con declarar a CONCAT como asociativa a izquierda y de mayor precedencia que la división, pero al parecer para que *Yacc* pueda resolver conflictos comparando orden de precedencias de símbolos con órdenes de precedencias de producciones hace falta cierta completitud sobre las declaraciones de precedencia de los símbolos que puedan llegar a ser el token corriente al decidir si reducir o no usando una producción.

- **y**: ubicación en el eje y (la esquina `(0,0)` se corresponde con la esquina superior izquierda del buffer) de una expresión *(heredado)*
- **svg**: output en términos de tags SVG generados por la expresión *(sintetizado)*

La manera de hacer pasadas sobre el AST se corresponde a la metodología del patrón de diseño "*Visitors*" comunmente usado para iterar este tipo de estructuras y que permite usar polimorfismo y double-dispatch para hacer llamados recursivos sobre cada nodo.

2.4.1. Cálculo de atributos

A continuación explicamos coloquialmente cómo fuimos generando los atributos para decorar el AST.

- El atributo **"e"** correspondiente a la escala, se inicia en 1 sobre la raíz y se reduce a un 70 % de su valor en cada indexación. Se trata de un atributo heredado.
- El atributo **"a"** correspondiente al ancho es:
  - el 60 % de la escala para caracteres
  - suma de anchos de subexpresiones el caso de concatenaciones
  - suma del ancho de la expresión principal y máximo ancho entre los índices para indexaciones
  - máximo entre ancho del numerador y del denominador para divisiones
  - suma de la expresión mas ancho de los dos paréntesis para dicho caso
- El atributo **"h1"** lo sintetizamos con los siguientes valores:
  - el interlineado para caracteres
  - máximo entre ambos $h1$ para concatenaciones
  - máximo entre el $h1$ de la expresión principal y $h1$ del superíndice mas su altura desde la base de la expresión general
  - $h1$ del numerador menos el corrimiento para arriba desde la base de la línea de división para tales expresiones
  - $h1$ de la expresión principal para paréntesis
- El atributo **"h2"** es:
  - 0 para caracteres
  - máximo entre ambos $h2$ para concatenaciones
  - máximo entre $h2$ de la expresión principal y el $h2$ del subíndice mas su desfasaje con la base de la expresión principal considerando tambien que las letras miden el 70 % del interlineado desde el y hasta su base real. [4]

[4] La idea de considerar el corrimiento real desde la base del caracter y no del interlineado es que no se produzcan paddings fantasma en las divisiones (que corresponderían al espacio que queda entre la base del caracter y de su interlineado). Para paréntesis por ejemplo es 80 %. Se podría ir ajustando el porcentaje de acuerdo a cada caracter, pero como nos pareció que el padding generado

- $h2$ del denominador mas el corrimiento para arriba desde la base de la línea de división para las divisiones
- $h2$ de la expresión principal para paréntesis

- El atributo **"h"** se calcula como:
  - $h1$ para caracteres
  - h de la expresión principal para paréntesis
  - suma de $h1$ y $h2$ para las demás operaciones
- El atributo **"x"** se inicializa en 0 en la raíz y lo heredamos en cada caso como:
  - el mismo x pasado como parámetro para caracteres
  - el mismo x pasado como parámetro para la primer expresión de una concatenación y el x parámetro sumado al ancho de la primer expresión para la segunda
  - el mismo x pasado por parametro más la mitad de la diferencia entre el máximo de los dos anchos y el ancho del numerador o denominador según el caso, de modo que ambas expresiones queden verticalmente centradas
  - el mismo x argumento para las expresiones principales y ese mismo x mas el ancho de la expresión principal para los índices en las indexaciones
  - x argumento mas el ancho de un paréntesis para la expresión principal para los paréntesis
- El atributo **"y"** tambien se inicializa en 0 en la raíz y lo heredamos como:
  - el mismo y subido $h2$ del numerador para él mismo y bajado $h1$ del denominador para este, ambos casos con un desplazamiento hacia arriba por el corrimiento de la barra de división
  - el mismo y argumento bajado un cuarto del h de la expresión total para subíndices mientras que nuevamente el mismo y subido $0{,}45$ veces el $h1$ de la expresión principal para superíndices
  - para los paréntesis, caracteres y concatenación se mantiene igual la altura
- El atributo **"svg"** se toma concatenando los strings de las subexpresiones:
  - para caracteres se devuelve un tag de tipo text con 'x', 'y' y 'font-size' correspondientes a los atributos x, y y e del mismo
  - para divisiones se concatenan los SVG del numerador y denominador con una barra horizontal de longitud del ancho de la división a casi la mitad del interlineado de altura
  - para paréntesis se concatenan tags de texto con caracteres '(' y ')' escalados para ocupar el h de la expresión [5]

por esta diferencia era casi despreciable además de que se espera que las expresiones sean principalmente letras y caracteres aritméticos decidimos estandarizar el porcentaje de las letras.

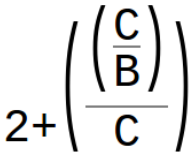
[5] Como dijimos antes, los paréntesis ocupan un 80 % de longitud del interlineado por lo que hay un 20 % de espacio vacío que, al escalarse, tambien se multiplica por lo que se lo restamos de modo que solamente quede un 20 % del interlineado como tal espacio.

3. Ejemplos

3.1. Expresiones válidas del lenguaje

1. `2+({{({C/B})}/C})`

   Output:

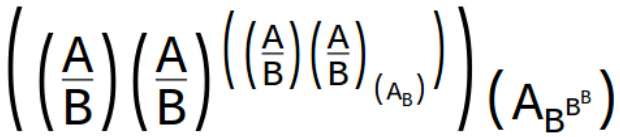
   

   AST generado:

   ```
   Concat(Concat(Chr(2),Chr(+)),GroupedPar(DivExpr(GroupedPar(DivExpr(Chr(C),Chr(B))),Chr(C))))
   ```

2. `({(A/B)(A/B)}^{((A/B)(A/B)_{(A_B)})})(A_{B^{B^{B}}})`

   Output:

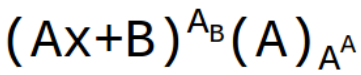
   

   AST generado:

   ```
   Concat(GroupedPar(SuperSub(Concat(GroupedPar(DivExpr(Chr(A),Chr(B))),GroupedPar(DivExpr(Chr(A),Chr(B)))),GroupedPar(Concat(GroupedPar(DivExpr(Chr(A),Chr(B))),SubSuper(GroupedPar(DivExpr(Chr(A),Chr(B))),LambdaExpr,GroupedPar(SubSuper(Chr(A),LambdaExpr,Chr(B))))),LambdaExpr)),GroupedPar(SubSuper(Chr(A),LambdaExpr,SuperSub(Chr(B),SuperSub(Chr(B),Chr(B),LambdaExpr),LambdaExpr))))
   ```

3. **`(Ax+B)^{A_B}(A)_{A^A}`**

   Output:

   AST generado:

   ```
   Concat(SuperSub(GroupedPar(Concat(Concat(Concat(Chr(A),Chr(x)),Chr(+)),Chr(B))),SubSuper(Chr(A),LambdaExpr,Chr(B)),LambdaExpr),SubSuper(GroupedPar(Chr(A)),LambdaExpr,SuperSub(Chr(A),Chr(A),LambdaExpr)))
   ```

4. `{a^2_1+a^2_2+...+a^2_n}/{||(a_1,...,a_n)||}`

   Output:

$$\frac{a_1^2+a_2^2+...+a_n^2}{||(a_1,...,a_n)||}$$

AST generado:

```
DivExpr(Concat(Concat(Concat(Concat(Concat(Concat(Concat(Concat(SuperSub(Chr(a),Chr(2),SubSuffix(Chr(1))),Chr(+)),SuperSub(Chr(a),Chr(2),SubSuffix(Chr(2)))),Chr(+)),Chr(.)),Chr(.)),Chr(.)),Chr(+)),SuperSub(Chr(a),Chr(2),SubSuffix(Chr(n)))),Concat(Concat(Concat(Concat(Chr(|),Chr(|)),GroupedPar(Concat(Concat(Concat(Concat(Concat(SubSuper(Chr(a),LambdaExpr,Chr(1)),Chr(,)),Chr(.)),Chr(.)),Chr(.)),Chr(,)),SubSuper(Chr(a),LambdaExpr,Chr(n))))),Chr(|)),Chr(|)))
```

5. `{{A^{A^{A^{A}}}_n}/{B^{B^{B}}_m}}+A+B`

   Output:

$$\frac{A_n^{A^{A^A}}}{B_m^{B^B}}+A+B$$

   AST generado:

```
Concat(Concat(Concat(Concat(DivExpr(SuperSub(Chr(A),SuperSub(Chr(A),SuperSub(Chr(A),Chr(A),LambdaExpr),LambdaExpr),SubSuffix(Chr(n))),SuperSub(Chr(B),SuperSub(Chr(B),Chr(B),LambdaExpr),SubSuffix(Chr(m)))),Chr(+)),Chr(A)),Chr(+)),Chr(B))
```

6. **Full**

   Output:

$$2+\left(\frac{\left(\frac{C}{B}\right)}{C}\right)+\left(\left(\frac{A}{B}\right)\left(\frac{A}{B}\right)^{\left(\left(\frac{A}{B}\right)\left(\frac{A}{B}\right)_{(A_B)}\right)}\right)(A_{B^{B^B}})+(Ax+B)^{A_B}(A)+A^A+\frac{a_1^2+a_2^2+...+a_n^2}{||(a_1,...,a_n)||}+\frac{A_n^{A^{A^A}}}{B_m^{B^B}}+A+B$$

3.2. Expresiones inválidas del lenguaje

1. `2+({}` (se abre un paréntesis que nunca se cierra).
2. `(A/B})` (se cierra una llave que nunca se abre).
3. `A^{b+1}_` (subíndice vacío)
4. `A4{}^` (superíndice vacío)
5. (cadena vacía)

6. `A^A^A` (asociatividad de superíndices)

7. `A^B_C^D` (> 1 superíndices en el mismo "scope" de paréntesis/llaves $\Rightarrow$ asociatividad implícita de superíndices)

4. Conclusiones

Valoramos la experiencia de haber podido aplicar nuestros conocimientos de técnicas de parsing (en este caso en particular de parsing *LALR*) y síntesis de atributos para entender con bastante detalle qué sucedía cuando generamos parsers con *Yacc*.

Gracias a estos conocimientos pudimos resolver y entender tanto la tabla generada como sus conflictos *Shift/Reduce* y *Reduce* a través de órdenes de precedencia y modificando fuertemente la gramática original para que tenga características *LALR*.
Luego, fuimos capaces de generar el AST de una forma que nos sirva para generar atributos.

Si bien esta generación de atributos para formar el SVG requirió ajustes manuales, esto nos permitió obtener resultados visualmente aceptables, con una variedad muy grande de fórmulas del lenguaje. Parte de la necesidad de los ajustes se debió a no tener en claro como lograr ciertos detalles en SVG, y exactamente que buscábamos generar. Aún así, los resultados fueron muy exitosos.

5. Bibliografía

Referencias

[1] **PLY**: Python LEX-Yacc http://www.dabeaz.com/ply/

6. Modo de Uso

Para utilizar el programa (requiere *PLY* para *Python 3*) usar:
`python3 main.py ‘‘EXPRESION’’ -o archivo.svg -p`

El parámetro -o es opcional, por default escribe al archivo “output.svg”.
El parámetro -p es opcional, sirve para imprimir por pantalla el svg.

Make tests genera los ejemplos que exhibimos en este informe.
make parsertests corre tests sobre el parser y la generación del AST.

Para probar el análisis lexicográfico sobre tokens:
`python3 lexer.py ‘‘EXPRESION’’`

7. Código fuente

Se omite el launcher del lexer dado que solamente es llamar al lexer desde un main y no forma parte del programa principal.

7.1. tokrules.py

```
        import ply.lex as lex

        tokens = (
            'CHR',
            'DIVIDE'
        )

        # los literales son chars que matchean de una
        literals = "_^(){}"

        def t_CHR(t):
            # el primer ^ toma complemento de los simbolos que siguen
            r'[^ _ \^ / \( \)\{\}]'
            return t

        def t_DIVIDE(t):
            r'/'
            return t

        t_ignore  = '\t'

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


```

7.2. AST.py

```
        class Expr: pass

        class LambdaExpr(Expr):
            def __init__(self):
                pass

            def __eq__(self, other):
                return isinstance(other, LambdaExpr)

            def __str__(self):
                return "LambdaExpr"

            def accept(self, visitor):
```

```
14                visitor.visitLambda(self)
15
16        # a
17        class Chr(Expr):
18
19            non_character_error_description = "Character should have length one"
20
21            def __init__(self, character):
22                if len(character) != 1:
23                    raise ValueError(Chr.non_character_error_description)
24                self.character = character
25
26            def __eq__(self, other):
27                if isinstance(other, Chr):
28                    return other.character == self.character
29                else:
30                    return False
31
32            def __str__(self):
33                return "Chr({0})".format(self.character)
34
35            def accept(self, visitor):
36                visitor.visitChr(self)
37
38        class DivExpr(Expr):
39            def __init__(self, leftExpr, rightExpr):
40                self.leftExpr = leftExpr
41                self.rightExpr = rightExpr
42
43            def __eq__(self, other):
44                if not isinstance(other, DivExpr):
45                    return False
46
47                return self.leftExpr == other.leftExpr and self.rightExpr == other.rightExpr
48            def __str__(self):
49                return "DivExpr({0},{1})".format(self.leftExpr, self.rightExpr)
50
51            def accept(self, visitor):
52                visitor.visitDiv(self)
53
54        # A B
55        class Concat(Expr):
56            def __init__(self, leftExpression, rightExpression):
57                self.leftExpression = leftExpression
58                self.rightExpression = rightExpression
59
60            def __eq__(self, other):
61                if not isinstance(other, Concat):
62                    return False
63                return self.leftExpression == other.leftExpression and self.rightExpression ==
```

```
    other.rightExpression
64
65          def __str__(self):
66              return "Concat({0},{1})".format(self.leftExpression, self.rightExpression)
67
68          def accept(self, visitor):
69              visitor.visitConcat(self)
70
71      class SuperSub(Expr):
72          def __init__(self, mainExpr, superExpr, subExpr):
73              self.mainExpr = mainExpr
74              self.superExpr = superExpr
75              self.subExpr = subExpr
76
77          def __eq__(self, other):
78              if not isinstance(other, SuperSub):
79                  return False
80
81              comp = True
82              comp = comp and (self.mainExpr == other.mainExpr)
83              comp = comp and (self.superExpr == other.superExpr)
84              comp = comp and (self.subExpr == other.subExpr)
85
86              return comp
87
88          def __str__(self):
89              return "SuperSub({0}, {1}, {2})".format(str(self.mainExpr), str(self.superExpr
    ), str(self.subExpr))
90
91          def accept(self, visitor):
92              visitor.visitSuperSub(self)
93
94      class SubSuper(Expr):
95          def __init__(self, mainExpr, subExpr, superExpr):
96              self.mainExpr = mainExpr
97              self.superExpr = superExpr
98              self.subExpr = subExpr
99
100         def __eq__(self, other):
101             if not isinstance(other, SubSuper):
102                 return False
103
104             comp = True
105             comp = comp and (self.mainExpr == other.mainExpr)
106             comp = comp and (self.superExpr == other.superExpr)
107             comp = comp and (self.subExpr == other.subExpr)
108
109             return comp
110
111         def __str__(self):
```

```
112                 return "SubSuper({0}, {1}, {2})".format(str(self.mainExpr), str(self.superExpr
    ), str(self.subExpr))
113 
114             def accept(self, visitor):
115                 visitor.visitSubSuper(self)
116 
117         class SuperSuffix(Expr):
118             def __init__(self, expr):
119                 self.expr = expr
120 
121             def __eq__(self, other):
122                 if not isinstance(other, SuperSuffix):
123                     return False
124                 return other.expr == self.expr
125 
126             def __str__(self):
127                 return "SuperSuffix({0})".format(self.expr)
128 
129             def accept(self, visitor):
130                 visitor.visitSuperSuffix(self)
131 
132         class SubSuffix(Expr):
133             def __init__(self, expr):
134                 self.expr = expr
135 
136             def __eq__(self, other):
137                 if not isinstance(other, SubSuffix):
138                     return False
139                 return other.expr == self.expr
140 
141             def __str__(self):
142                 return "SubSuffix({0})".format(self.expr)
143 
144             def accept(self, visitor):
145                 visitor.visitSubSuffix(self)
146 
147         # ()
148         class GroupedPar(Expr):
149             def __init__(self, expr):
150                 self.expr = expr
151 
152             def __eq__(self, other):
153                 if not isinstance(other, GroupedPar):
154                     return False
155                 return self.expr == other.expr
156 
157             def __str__(self):
158                 return "GroupedPar({0})".format(self.expr)
159 
160             def accept(self, visitor):
```

```
161             visitor.visitGroupedPar(self)
162
163
```

7.3. parser_rules.py

```
1       import ply.yacc as yacc
2       from tokrules import tokens
3       from AST import *
4
5       SYNTAX_ERROR_IN_INPUT_ERROR_MESSAGE = "Syntax error in input!"
6
7       precedence = (
8           ('left', 'DIV'),
9
10          ('left', '{', '('),
11
12          ('left', 'CHR'),
13
14          ('left', 'CONCAT'),
15
16          ('nonassoc', '^'),
17              ('nonassoc', '_'),
18
19      )
20
21      def p_unary(p):
22          '''expression : unary_exp'''
23          p[0] = p[1]
24
25      def p_expression_chr(p):
26          '''unary_exp : CHR'''
27          p[0] = Chr(p[1])
28
29      def p_expression_concat(p):
30          '''expression : expression expression %prec CONCAT'''
31          # %prec asocia la precedencia de la produccion a la del pseudosimbolo CONCAT
32          p[0] = Concat(p[1], p[2])
33
34      def p_expression_div(p):
35          '''expression : expression DIVIDE expression %prec DIV'''
36          p[0] = DivExpr(p[1], p[3])
37
38      def p_expression_super(p):
39          '''expression : unary_exp '^' unary_exp subexp'''
40          p[0] = SuperSub(p[1], p[3], p[4])
41
42      def p_expression_sub(p):
43          '''expression : unary_exp '_' unary_exp superexp'''
44          p[0] = SubSuper(p[1], p[3], p[4])
```

```
45
46      def p_superexp_lambda(p):
47          '''superexp : lambda'''
48          p[0] = LambdaExpr()
49
50      def p_superexp_expr(p):
51          '''superexp : '^' unary_exp'''
52          p[0] = SuperSuffix(p[2])
53
54      def p_subexp_lambda(p):
55          '''subexp : lambda'''
56          p[0] = LambdaExpr()
57
58      def p_subexp_expr(p):
59          '''subexp : '_' unary_exp'''
60          p[0] = SubSuffix(p[2])
61
62      def p_expression_grouped_par(p):
63          '''unary_exp : '(' expression ')' '''
64          p[0] = GroupedPar(p[2])
65
66      def p_expression_grouped_brkt(p):
67          '''unary_exp : '{' expression '}' '''
68          # Curly Brackets no aparecen en el AST
69          p[0] = p[2]
70
71      def p_lambda(p):
72          '''lambda :'''
73
74      def p_error(p):
75          raise ValueError(SYNTAX_ERROR_IN_INPUT_ERROR_MESSAGE)
76
77
```

7.4. AST_visitors.py

```
1       from AST import *
2       from copy import copy
3
4       class Visitor: pass
5
6       class EscaleVisitor(Visitor):
7
8           def __init__(self, escale):
9               self.e = escale
10
11          def visitLambda(self, expr):
12              pass
13
14          def visitChr(self, expr):
```

```
15              expr.e = self.e
16
17          def visitDiv(self, expr):
18              expr.e = self.e
19              expr.leftExpr.accept(self)
20              expr.rightExpr.accept(self)
21
22          def visitConcat(self, expr):
23              expr.e = self.e
24              expr.leftExpression.accept(self)
25              expr.rightExpression.accept(self)
26
27          def visitSubSuper(self, expr):
28              expr.e = self.e
29              expr.mainExpr.accept(self)
30              expr.superExpr.accept(EscaleVisitor(0.7*self.e))
31              expr.subExpr.accept(EscaleVisitor(0.7*self.e))
32
33          def visitSuperSub(self, expr):
34              expr.e = self.e
35              expr.mainExpr.accept(self)
36              expr.superExpr.accept(EscaleVisitor(0.7*self.e))
37              expr.subExpr.accept(EscaleVisitor(0.7*self.e))
38
39          def visitSuperSuffix(self, supersuffix_expr):
40              supersuffix_expr.e = self.e
41              supersuffix_expr.expr.accept(self)
42
43          def visitSubSuffix(self, subsuffix_expr):
44              subsuffix_expr.e = self.e
45              subsuffix_expr.expr.accept(self)
46
47          def visitGroupedPar(self, grouped_expr):
48              grouped_expr.e = self.e
49              grouped_expr.expr.accept(self)
50
51      class WidthVisitor(Visitor):
52
53          def __init__(self): pass
54
55          def visitLambda(self, expr):
56              expr.a = 0
57
58          def visitChr(self, expr):
59              expr.a = 0.6 * expr.e
60
61          def visitDiv(self, expr):
62              expr.leftExpr.accept(self)
63              expr.rightExpr.accept(self)
64              expr.a = max(expr.leftExpr.a, expr.rightExpr.a)
```

```
65
66          def visitConcat(self, expr):
67              expr.leftExpression.accept(self)
68              expr.rightExpression.accept(self)
69              expr.a = expr.leftExpression.a + expr.rightExpression.a
70
71          def visitSubSuper(self, expr):
72              expr.mainExpr.accept(self)
73              expr.superExpr.accept(self)
74              expr.subExpr.accept(self)
75              expr.a = expr.mainExpr.a + max(expr.superExpr.a, expr.subExpr.a)
76
77          def visitSuperSub(self, expr):
78              expr.mainExpr.accept(self)
79              expr.superExpr.accept(self)
80              expr.subExpr.accept(self)
81              expr.a = expr.mainExpr.a + max(expr.superExpr.a, expr.subExpr.a)
82
83          def visitSuperSuffix(self, supersuffix_expr):
84              supersuffix_expr.expr.accept(self)
85              supersuffix_expr.a = supersuffix_expr.expr.a
86
87          def visitSubSuffix(self, subsuffix_expr):
88              subsuffix_expr.expr.accept(self)
89              subsuffix_expr.a = subsuffix_expr.expr.a
90
91          def visitGroupedPar(self, grouped_expr):
92              grouped_expr.expr.accept(self)
93              grouped_expr.a = grouped_expr.expr.a + grouped_expr.e * 2 * 0.6 # contar ()
94
95      class HVisitor(Visitor):
96
97          def __init__(self): pass
98
99          def visitLambda(self, expr):
100             expr.h1 = 0
101             expr.h2 = 0
102             expr.h = 0
103
104         def visitChr(self, expr):
105             expr.h1 = expr.e
106             expr.h2 = 0
107             expr.h = expr.e
108
109         def visitDiv(self, expr):
110             expr.leftExpr.accept(self)
111             expr.rightExpr.accept(self)
112             expr.h1 = expr.leftExpr.h1 + expr.leftExpr.h2 + expr.e*0.6
113             expr.h2 = expr.rightExpr.h1 + expr.rightExpr.h2 - expr.e*0.6
114             expr.h = expr.h1 + expr.h2
```

```
115
116         def visitConcat(self, expr):
117             expr.leftExpression.accept(self)
118             expr.rightExpression.accept(self)
119             expr.h1 = max(expr.leftExpression.h1, expr.rightExpression.h1)
120             expr.h2 = max(expr.leftExpression.h2, expr.rightExpression.h2)
121             expr.h = expr.h1 + expr.h2
122
123         def visitSubSuper(self, expr):
124             expr.mainExpr.accept(self)
125             expr.superExpr.accept(self)
126             expr.subExpr.accept(self)
127
128             super_h1 = 0 if isinstance(expr.superExpr, LambdaExpr) else expr.mainExpr.h1
    *0.45 + expr.mainExpr.e + expr.superExpr.h1 - expr.superExpr.e
129             #sumamos el h1 hasta el 'y' del superindice con su h1 y le restamos su escala
    (que sino se suma dos veces)
130
131             expr.h1 = max(expr.mainExpr.h1, super_h1)
132
133             sub_h2 = expr.subExpr.h2 + expr.subExpr.e + expr.mainExpr.h*0.25 - expr.
    mainExpr.e*0.7 # en un dibujo se ve bien, notar el 0.7 porque los char del mainExpr no
    miden toda la escala de largo sino que solo el 70% y el restante es espacio vacio
134             expr.h2 = max(expr.mainExpr.h2, sub_h2)
135             expr.h = expr.h1 + expr.h2
136
137         def visitSuperSub(self, expr):
138             expr.mainExpr.accept(self)
139             expr.superExpr.accept(self)
140             expr.subExpr.accept(self)
141
142             sub_h2 = 0 if isinstance(expr.subExpr, LambdaExpr) else expr.subExpr.h2 + expr
    .subExpr.e + expr.mainExpr.h*0.25 - expr.mainExpr.e*0.7
143
144             expr.h2 = max(expr.mainExpr.h2, sub_h2)
145             super_h1 = expr.mainExpr.h1*0.45 + expr.mainExpr.e + expr.superExpr.h1 - expr.
    superExpr.e
146             expr.h1 = max(expr.mainExpr.h1, super_h1)
147             expr.h = expr.h1 + expr.h2
148
149         def visitSuperSuffix(self, supersuffix_expr):
150             supersuffix_expr.expr.accept(self)
151             supersuffix_expr.h1 = supersuffix_expr.expr.h1
152             supersuffix_expr.h2 = supersuffix_expr.expr.h2
153             supersuffix_expr.h = supersuffix_expr.h1 + supersuffix_expr.h2
154
155         def visitSubSuffix(self, subsuffix_expr):
156             subsuffix_expr.expr.accept(self)
157             subsuffix_expr.h1 = subsuffix_expr.expr.h1
158             subsuffix_expr.h2 = subsuffix_expr.expr.h2
```

```
159                 subsuffix_expr.h = subsuffix_expr.h1 + subsuffix_expr.h2
160
161             def visitGroupedPar(self, grouped_expr):
162                 grouped_expr.expr.accept(self)
163                 grouped_expr.h1 = grouped_expr.expr.h1
164                 grouped_expr.h2 = grouped_expr.expr.h2
165                 grouped_expr.h = grouped_expr.expr.h
166
167         class XVisitor(Visitor):
168
169             def __init__(self, pos):
170                 self.pos = pos
171
172             def visitLambda(self, expr): pass
173
174             def visitChr(self, expr):
175                 expr.x = self.pos
176
177             def visitDiv(self, expr):
178                 divwidth = max(expr.leftExpr.a, expr.rightExpr.a)
179                 #ambos centrados respecto de la linea de division
180                 expr.leftExpr.accept(XVisitor(self.pos + (divwidth - expr.leftExpr.a)/2))
181                 expr.rightExpr.accept(XVisitor(self.pos + (divwidth - expr.rightExpr.a)/2))
182                 expr.x = self.pos
183
184             def visitConcat(self, expr):
185                 expr.leftExpression.accept(self)
186                 expr.rightExpression.accept(XVisitor(self.pos + expr.leftExpression.a))
187                 expr.x = self.pos
188
189             def visitSubSuper(self, expr):
190                 expr.mainExpr.accept(self)
191                 expr.subExpr.accept(XVisitor(self.pos + expr.mainExpr.a))
192                 expr.superExpr.accept(XVisitor(self.pos + expr.mainExpr.a))
193                 expr.x = self.pos
194
195             def visitSuperSub(self, expr):
196                 expr.mainExpr.accept(self)
197                 expr.superExpr.accept(XVisitor(self.pos + expr.mainExpr.a))
198                 expr.subExpr.accept(XVisitor(self.pos + expr.mainExpr.a))
199                 expr.x = self.pos
200
201             def visitSuperSuffix(self, supersuffix_expr):
202                 supersuffix_expr.expr.accept(self)
203                 supersuffix_expr.x = self.pos
204
205             def visitSubSuffix(self, subsuffix_expr):
206                 subsuffix_expr.expr.accept(self)
207                 subsuffix_expr.x = self.pos
208
```

```
209             def visitGroupedPar(self, grouped_expr):
210                 grouped_expr.expr.accept(XVisitor(self.pos + 0.6*grouped_expr.e)) # dejar
    espacio para '('
211                 grouped_expr.x = self.pos
212 
213         class YVisitor(Visitor):
214 
215             def __init__(self, pos):
216                 self.pos = pos
217 
218             def visitLambda(self, expr): pass
219 
220             def visitChr(self, expr):
221                 expr.y = self.pos
222 
223             def visitDiv(self, expr):
224                 expr.leftExpr.accept(YVisitor(self.pos - expr.leftExpr.h2 - expr.e*0.6))
225                 expr.rightExpr.accept(YVisitor(self.pos + expr.rightExpr.h1 - expr.e*0.6))
226                 expr.y = self.pos
227 
228             def visitConcat(self, expr):
229                 expr.leftExpression.accept(self)
230                 expr.rightExpression.accept(self)
231                 expr.y = self.pos
232 
233             def visitSubSuper(self, expr):
234                 expr.mainExpr.accept(self)
235                 expr.subExpr.accept(YVisitor(self.pos + expr.mainExpr.h*0.25)) #parece ser la
    formula que usaron en el ejemplo del enunciado
236                 expr.superExpr.accept(YVisitor(self.pos - expr.mainExpr.h1*0.45)) #visto en
    clase
237                 expr.y = self.pos
238 
239             def visitSuperSub(self, expr):
240                 expr.mainExpr.accept(self)
241                 expr.subExpr.accept(YVisitor(self.pos + expr.mainExpr.h*0.25))
242                 expr.superExpr.accept(YVisitor(self.pos - expr.mainExpr.h1*0.45))
243                 expr.y = self.pos
244 
245             def visitSuperSuffix(self, supersuffix_expr):
246                 supersuffix_expr.expr.accept(self)
247                 supersuffix_expr.y = self.pos
248 
249             def visitSubSuffix(self, subsuffix_expr):
250                 subsuffix_expr.expr.accept(self)
251                 subsuffix_expr.y = self.pos
252 
253             def visitGroupedPar(self, grouped_expr):
254                 if isinstance(grouped_expr.expr, DivExpr):
255                     grouped_expr.expr.accept(YVisitor(self.pos - 0.3 * grouped_expr.expr.e))
```

```
256                 else:
257                     grouped_expr.expr.accept(self)
258 
259                 grouped_expr.y = self.pos
260 
261         class SVGRendererVisitor(Visitor) :
262 
263             def __init__(self): pass
264 
265             def visitLambda(self, expr):
266                 expr.svg = ""
267 
268             def visitChr(self, expr):
269                 expr.svg = "<text x=\"{}\" y=\"{}\" font-size=\"{}\">{}</text> \n".format(expr
    .x, expr.y, expr.e, expr.character)
270 
271             def visitDiv(self, expr):
272                 expr.leftExpr.accept(self)
273                 expr.rightExpr.accept(self)
274                 expr.svg = expr.leftExpr.svg
275                 expr.svg += "<line x1=\"{}\" y1=\"{}\" x2=\"{}\" y2=\"{}\" stroke-width
    =\"0.03\" stroke=\"black\"/>".format(expr.x, expr.y-expr.e*0.45, expr.x + expr.a, expr.y-
    expr.e*0.45)
276                 expr.svg += expr.rightExpr.svg
277 
278             def visitConcat(self, expr):
279                 expr.leftExpression.accept(self)
280                 expr.rightExpression.accept(self)
281                 expr.svg = expr.leftExpression.svg + expr.rightExpression.svg
282 
283             def visitSubSuper(self, expr):
284                 expr.mainExpr.accept(self)
285                 expr.superExpr.accept(self)
286                 expr.subExpr.accept(self)
287                 expr.svg = expr.mainExpr.svg + expr.subExpr.svg + expr.superExpr.svg
288 
289             def visitSuperSub(self, expr):
290                 expr.mainExpr.accept(self)
291                 expr.superExpr.accept(self)
292                 expr.subExpr.accept(self)
293                 expr.svg = expr.mainExpr.svg + expr.superExpr.svg + expr.subExpr.svg
294 
295             def visitSuperSuffix(self, supersuffix_expr):
296                 supersuffix_expr.expr.accept(self)
297                 supersuffix_expr.svg = supersuffix_expr.expr.svg
298 
299             def visitSubSuffix(self, subsuffix_expr):
300                 subsuffix_expr.expr.accept(self)
301                 subsuffix_expr.svg = subsuffix_expr.expr.svg
302 
```

```
303         def visitGroupedPar(self, grouped_expr):
304             grouped_expr.expr.accept(self)
305             times_scale = grouped_expr.h/grouped_expr.e
306             grouped_expr.svg = "<text x=\"0\" y=\"0\" font-size=\"{}\" transform=\"
    translate({},{}) scale(1,{})\">(</text> \n".format(grouped_expr.e, grouped_expr.x,
    grouped_expr.y-(times_scale-1)*0.2*grouped_expr.e, times_scale)
307
308             grouped_expr.svg += grouped_expr.expr.svg
309
310             grouped_expr.svg += "<text x=\"0\" y=\"0\" font-size=\"{}\" transform=\"
    translate({},{}) scale(1,{})\">)</text> \n".format(grouped_expr.e, grouped_expr.x +
    grouped_expr.a - 0.6*grouped_expr.e, grouped_expr.y-(times_scale-1)*0.2*grouped_expr.e,
    times_scale)
311
312
```

7.5. ParserTest.py

```
1       import unittest
2       import main
3       from AST import *
4       from AST_visitors import *
5
6       class ParserTest(unittest.TestCase):
7           def test_cant_create_chr_expression_with_non_character(self):
8               with self.assertRaises(ValueError) as contextManager:
9                   chr = Chr("")
10              self.assertEqual(str(contextManager.exception), Chr.
    non_character_error_description)
11
12          def test_character_is_chr_expression(self):
13              self.assertEqual(main.ast_generate("a"), Chr('a'))
14
15          def test_double_underscore_raises_syntax_error(self):
16              self.assert_raises_syntax_error("a_a_a")
17
18          def assert_raises_syntax_error(self, a):
19              with self.assertRaises(ValueError) as cm:
20                  main.ast_generate(a)
21              self.assertEqual(str(cm.exception), "Syntax error in input!")
22
23          def test_double_superscript_raises_syntax_error(self):
24              self.assert_raises_syntax_error("a^a^a")
25
26          def test_superscript(self):
27              ast = SuperSub(Chr('a'), Chr('b'), LambdaExpr())
28              superScriptStr = "a^b"
29              self.assert_equal_ast(ast, superScriptStr)
30
31          def assert_equal_ast(self, ast, superScriptStr):
```

```
32              parsedAst = main.ast_generate(superScriptStr)
33              self.assertEqual(parsedAst, ast)
34
35          def test_subscript(self):
36              ast = SubSuper(Chr('a'), Chr('b'), LambdaExpr())
37              self.assert_equal_ast(ast, "a_b")
38
39          def test_subscript_and_superscript(self):
40              ast = SubSuper(Chr('a'), Chr('b'), SuperSuffix(Chr('c')))
41              self.assert_equal_ast(ast, "a_b^c")
42
43          def test_superscript_and_subscript(self):
44              ast = SuperSub(Chr('a'), Chr('b'), SubSuffix(Chr('c')))
45              self.assert_equal_ast(ast, "a^b_c")
46
47          def test_qonda(self):
48              #Esto deberia fallar no?
49              self.assert_raises_syntax_error("a^b_c^d")
50              self.assert_raises_syntax_error("a_b^c_d")
51
52          def test_div(self):
53              ast = DivExpr(Chr('a'), Chr('b'))
54              self.assert_equal_ast(ast, "a/b")
55
56          def test_div_minus(self):
57              ast = DivExpr(Chr('a'), Concat(Concat(Chr('b'), Chr('-')),Chr('c')))
58              self.assert_equal_ast(ast, "a/b-c")
59
60          def test_parse_complex_formula(self):
61              leftDivAst = Concat(SuperSub(Chr('A'), Chr('B'), LambdaExpr()), SuperSub(Chr('
    C'), Chr('D'), LambdaExpr()))
62              rightDivAst = Concat(Concat(SuperSub(Chr('E'), Chr('F'), SubSuffix(Chr('G'))),
     Chr('+')), Chr('H'))
63              parentesisedAst = GroupedPar(DivExpr(leftDivAst, rightDivAst))
64              ast = Concat(Concat(parentesisedAst, Chr('-')), Chr('I'))
65
66              self.assert_equal_ast(ast, "(A^BC^D/E^F_G+H)-I")
67
68          def test_complex_formula_equal_formula_with_curly_brackets(self):
69              astComplexFormula = main.ast_generate("(A^BC^D/E^F_G+H)-I")
70              curlyBracketsFormula = "{({{A^B}{C^D}}/{{{E^F_G}+}H})-}I"
71              astCurlyBrackets = main.ast_generate(curlyBracketsFormula)
72              self.assertEqual(astComplexFormula, astCurlyBrackets)
73
74      if __name__ == '__main__':
75          unittest.main()
76
77
```

7.6. main.py

```
#!/usr/bin/python3
from sys import argv, exit
import argparse

from ply.lex import lex
import tokrules

from ply.yacc import yacc
import parser_rules

import AST_visitors

def ast_generate(input_str):
    lexer = lex(module=tokrules)
    parser = yacc(module=parser_rules)
    ast = parser.parse(input_str, lexer)
    return ast

def generar(input):
    ast = ast_generate(input)
    # print(ast)
    ast.accept(AST_visitors.EscaleVisitor(1))
    ast.accept(AST_visitors.WidthVisitor())
    ast.accept(AST_visitors.HVisitor())
    ast.accept(AST_visitors.XVisitor(0))
    ast.accept(AST_visitors.YVisitor(ast.h1))
    ast.accept(AST_visitors.SVGRendererVisitor())

    return ast

if __name__ == "__main__":
    argparser = argparse.ArgumentParser(description='Parsea y genera SVG. Lo podes mandar a un archivo')
    argparser.add_argument('expression', type=str,
                           help='Expresion para parsear')

    argparser.add_argument('--output', '-o', dest='output_filename', default='output.svg', type=str,
                           help="Archivo output svg")

    argparser.add_argument('--print-svg', '-p', dest='print_svg', action='store_const',
                           const=True, default=False,
                           help='Imprimir svg')

    args = argparser.parse_args()

    ast_with_attributes = generar(args.expression)
```

```
46              result = "<svg xmlns=\"http://www.w3.org/2000/svg\" width=\"{}\" height=\"{}\"
    version=\"1.1\" style=\"background: white\">\n<g transform=\"scale(40) translate(1,1)\"
    font-family=\"Courier\">\n".format(ast_with_attributes.a*50+80, (ast_with_attributes.h1+
    ast_with_attributes.h2)*40+80, ast_with_attributes.h1)
47              result += ast_with_attributes.svg
48              result += "</g>\n</svg>\n"
49
50              output_file = open(args.output_filename, "w")
51              output_file.write(result)
52              output_file.close()
53
54              if args.print_svg:
55                  print(result)
56
57
```