Teoria de Lenguajes

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

TP1

Dibu: Graficos vectoriales para niños
December 1, 2016

Grupo: Parseamela Gramatica

Integrante	LU	Correo electrónico
Mauro Cherubini	835/13	cheru.mf@gmail.com
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Introduccion

El presente documento describe el lenguaje Dibu, un lenguaje de graficos vectoriales para niños. Para construirlo, a partir del diseno del lenguaje se procedio a construir su gramatica, para luego programar su correspondiente Lexer y Parser. Toda la implementacion fue realizada utilizando Python y la libreria PLY (Python Lex-Yacc) [1].

2 Gramatica

Al momento de disenar la gramatica de Dibu, tuvimos que tener en cuenta varias caracteristicas del lenguaje:

1. El atributo size, que define el tamano de un canvas SVG, no necesariamente es la primera instruccion ni necesariamente existe.
2. Para toda funcion, sus parametros deben ser conmutativos.
3. Los argumentos opcionales se pueden introducir a las funciones de dos maneras diferentes, por medio de atributos normales o por medio de atributos de estilo.
4. Los literales que forman parte del lenguaje son:
   - Numeros (enteros o de punto flotante)
   - Strings
   - Pares/Puntos
   - Arreglos de literales
5. size puede ser un identificador de funcion o un atributo.
6. Las instrucciones se pueden separar por medio de espacios o linebreaks.

La siguiente gramatica corresponde al lenguaje Dibu. La misma se encuentra en el lenguaje aceptado por PLY:

```
Rule 0     S' -> start
Rule 1     start -> expression
Rule 2     expression -> ID argument_list expression
Rule 3     expression -> <empty>
Rule 4     argument_list -> ID EQUALS type_value
Rule 5     argument_list -> argument_list COMMA ID EQUALS type_value
Rule 6     type_value -> variable
Rule 7     type_value -> QUOTATION_MARK argl_style QUOTATION_MARK
Rule 8     argl_style -> ID COLON style_var SEMICOLON
Rule 9     argl_style -> argl_style ID COLON style_var SEMICOLON
Rule 10    style_var -> ID
Rule 11    style_var -> NUMBER
Rule 12    variable -> NUMBER
Rule 13    variable -> STRING
Rule 14    variable -> LPAREN variable COMMA variable RPAREN
Rule 15    variable -> LBRACKET variable_list RBRACKET
Rule 16    variable_list -> variable
Rule 17    variable_list -> variable_list COMMA variable
```

[2]

[1] PLY (Python Lex-Yacc): http://www.dabeaz.com/ply/ply.html

[2] Intentamos hacer que start fuese el nodo de comienzo de la gramatica, pero PLY no estuvo de nuestro lado.

Los terminales y no terminales son:

```
Terminals, with rules where they appear

COLON                : 8 9
COMMA                : 5 14 17
EQUALS               : 4 5
ID                   : 2 4 5 8 9 10
LBRACKET             : 15
LPAREN               : 14
NUMBER               : 11 12
QUOTATION_MARK       : 7
RBRACKET             : 15
RPAREN               : 14
SEMICOLON            : 8 9
STRING               : 13
error                :

Nonterminals, with rules where they appear

argl_style           : 7 9
argument_list        : 2 5
expression           : 1 2
start                : 0
style_var            : 8 9
type_value           : 4 5
variable             : 6 14 14 16 17
variable_list        : 15 17
```

3 Lexer

En primer lugar, para identificar los tokens tuvimos un problema. Los atributos strings eran muy similares a los atributos de estilo. Por ejemplo:

```
fill="red"
style="fill: none;"
```

Por esta razon, decidimos que para distinguir entre estos dos tipos de strings usariamos el caracter :. El caracter : en general es solo valido para los atributos de estilo, por lo que parecio razonable.

A continuacion mostramos las expresiones regulares que utilizamos para identificar los diferentes tokens.

Token	Expresion Regular
:	:
,	,
=	=
ID	[_a-zA-Z0-9] *
(	(
)	)
NUMBER	[0-9]+(.[0-9]+)?
[	[
]	]
STRING	” [_a-zA-Z0-9]* ”

Table 1: Expresiones regulares para los tokens

4 Analisis Semantico

Nuestro lenguaje reconoce los siguientes tipos de errores semanticos:

1. Verifica si las funciones utilizadas son validas.
2. Verifica si el size del canvas ya fue definido, propagando un bool que dice si ya esta definido o no hacia abajo el AST.
3. Verifica si todos los atributos obligatorios de una clase fueron definidos.
4. Verifica si todos los atributos son validos, considerando atributos opcionales.
5. Verifica si ningun atributo se definio dos veces.
6. Verifica que los tipos de todos los literales son validos.

- Cada funcion tiene dos tipos de parametros asociados. Obligatorios y opcionales.

5 Ejemplos

5.1 aim.svg

```
size height=100, width=100
circle center=(50, 50), radius=50, fill="red"
circle center=(50, 50), radius=25, fill="black"
```

Figure 1: aim.svg

5.2 grid.svg

```
rectangle upper_left=(0,0), size=(200, 200), fill="yellow"
polygon points=[(0,0), (50, 50), (0, 100)], style="stroke: black; stroke-width: 3; fill: none;"
polygon points=[(0,0), (50, 50), (100, 0)], style="stroke: black; stroke-width: 3; fill: none;"
polygon points=[(0, 100), (50, 150), (0, 200)], style="stroke: black; stroke-width: 3; fill:none;"
polygon points=[(0, 200), (50, 150), (100, 200)], style="stroke: black; stroke-width: 3; fill: nor
polygon points=[(100, 200), (150, 150), (200, 200)], style="stroke: black; stroke-width: 3; fill:
polygon points=[(200, 200), (150, 150), (200, 100)], style="stroke: black; stroke-width: 3; fill:
polygon points=[(200, 100), (150, 50), (200, 0)], style="stroke: black; stroke-width: 3; fill: nor
polygon points=[(200, 0), (150, 50), (100, 0)], style="stroke: black; stroke-width: 3; fill:none;"
```

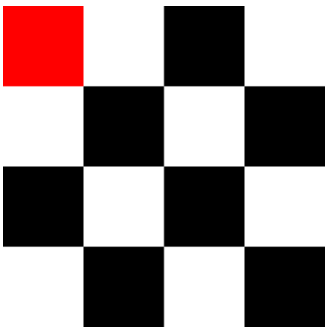

Figure 2: aim.svg

5.3 squares.svg

```
size height=200, width=200
rectangle upper_left=(0,0), size=(50, 50), fill="red"
rectangle upper_left=(100,0), size=(50, 50)
rectangle upper_left=(50,50), size=(50, 50)
rectangle upper_left=(150,50), size=(50, 50)
rectangle upper_left=(0,100), size=(50, 50)
rectangle upper_left=(100,100), size=(50, 50)
rectangle upper_left=(50,150), size=(50, 50)
rectangle upper_left=(150,150), size=(50, 50)
```

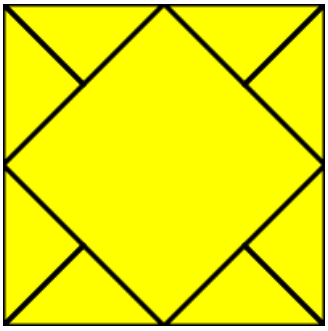

Figure 3: squares.svg

5.4 Deteccion de errores

5.4.1 Errores lexicos

lexical_error.dibu

```
size height=200, width=200
rectangle upper_left=(0,0), size=(50, 50), fill="red!
```

Output:

```
Lexical error at line 2, column 53:
rectangle upper_left=(0,0), size=(50, 50), fill="red!
                                                    ^
```

5.4.2 Errores sintacticos

syntax_error.dibu

```
size height=200, width=200
rectangle upper_left=(0,0), size=(50, 50), fill="red"
rectangle upper_left=(100,0), size=(50, 50)
rectangle upper_left=(50,50), size=(50, 50,
rectangle upper_left=(150,50), size=(50, 50)
```

Output:

```
Parse error at line 4, column 43:
rectangle upper_left=(50,50), size=(50, 50,
                                          ^
```

5.4.3 Errores semanticos

semantic_error.dibu

```
size height=200, width=200
rectangle upper_left=("red",0), size=(50, 50), fill="red"
```

Output:

```
Invalid attribute type for x in point.
```

6 Armado del XML del SVG

Una vez parseado el codigo, se debe traducir a SVG. Los SVG (Scalable Vector Graphics) son archivos XML con cierto formato. Para una funcion de Dibu, en general su nombre de funcion y atributo coincide con el identificador utilizado por SVG en XML. Sin embargo, en el caso de las funciones muchas veces esto no sucede. Por esta razon en la clase AvailableFunctions(), al agregar una funcion tambien se debe agregar su alias de SVG. Este es el caso de 'rectangle', donde en el SVG se la llama 'rect'.

En cuanto a los atributos, algo muy similar sucedio. En el caso de los puntos, los arreglos y los atributos de texto, los mismos tenian un formato muy particular, por lo que decidimos hardcodear el armado de estos atributos en la clase SVG.

Para el armado de los XML se utilizo la libreria ElementTree de Python.

7 Requerimientos

Para poder correr nuestro parser, se necesita Python 2 con las siguientes dependencias/paquetes:

1. PLY: Python Lex-Yacc
2. ElementTree: Viene en la libreria standard de Python, se utiliza para generar XMLs.
3. xml.dom.minidom: Utilizado para mejorar el aspecto visual de los XML generados.

Luego, para correr el parser simplemente hacer:

```
python parser.py dibu/aim.dibu dibu/aim.svg
```

Donde el primer parametro indica el path al script en formato Dibu y el segundo parametro el path al archivo donde escribir el SVG resultante.

8 Código fuente

8.1 lexer.py

```
import ply.lex as lex

class DibuLexer(object):

class LexicalException(Exception):
pass

# list of tokens
tokens = (
'ID',
'EQUALS',
'LPAREN',
'RPAREN',
'LBRACKET',
'RBRACKET',
'COMMA',
'NUMBER',
'STRING',
'QUOTATION_MARK',
'COLON',
'SEMICOLON'
)

# regular expression rules for simple ts
t_EQUALS = r"="
t_LPAREN = r"\("
t_RPAREN = r"\)"
t_LBRACKET = r"\["
t_RBRACKET = r"\]"
t_COMMA = r","
t_QUOTATION_MARK = r"\""
t_COLON = r":"
t_SEMICOLON = r";"
t_ignore = " \t\r"

def t_ID(self, t):
r"[\-_a-zA-Z0-9]*"
return t

def t_NUMBER(self, t):
r"[0-9]+(\.[0-9]+)?"
if t.value.find(".") >= 0:
t.value = float(t.value)
else:
t.value = int(t.value)
return t

def t_NEWLINE(self, t):
r"\n+"
t.lexer.lineno += len(t.value)

def t_STRING(self, t):
```

```
r"\"[ _a-zA-Z0-9]*\""
t.value = t.value[1:-1]
return t

def find_column(self, t):
last_cr = self.code.rfind('\n', 0, t.lexpos)
if last_cr < 0: last_cr = 0
column = (t.lexpos - last_cr) + (1 if last_cr == 0 else 0)
return column

def t_error(self, t):

line = t.lineno
column = self.find_column(t)
print 'Lexical error at line %d, column %d:' % (line, column)
print self.code.split('\n')[line - 1]
print ' ' * (column - 1) + '^'
raise self.LexicalException()

def dump_tokens(self, filename):
t = self.lexer.token()
output_file = open(filename, "w")

while t is not None:
output_file.write("type:" + t.type)
output_file.write(" value:" + str(t.value))
output_file.write(" line:" + str(t.lineno))
output_file.write(" position:" + str(t.lexpos))
output_file.write("\n")

t = self.lexer.token()

output_file.close()

def input(self, code):
self.code = code

def lex(self, code):
self.lexer.input(code)

def __init__(self, code):
self.code = code;
self.lexer = lex.lex(module=self)

if __name__ == "__main__":

code = "size height=30, width=200 text t=\"I love Dibu\", at=(0,15), fill=\"red\"";

m = DibuLexer(code)
m.lex(code)
m.dump_tokens("test.txt")
```

8.2 parser.py

```
from ply import yacc
from lexer import DibuLexer
```

```
from dibu import AvailableFunctions
from expressions import SemanticException

from expressions import *
from xml.dom.minidom import parseString as xmlParse
from sys import argv, exit

class DibuParser(object):

start = 'start'

tokens = DibuLexer.tokens

class ParseException(Exception):
pass

def p_start(self, subexpressions):
'start : expression'
subexpressions[0] = Start(subexpressions[1], self.availableFunctions)

def p_expression(self, subexpressions):
'expression : ID argument_list expression'
subexpressions[0] = ExpressionList(subexpressions[1], subexpressions[2], subexpressions[3], self.availableFun

def p_expression_empty(self, subexpressions):
'expression :'
subexpressions[0] = EmptyExpression();

def p_argument_list_single(self, subexpressions):
'argument_list : ID EQUALS type_value'
subexpressions[0] = ArgumentList(subexpressions[1], subexpressions[3])

def p_argument_list_append(self, subexpressions):
'argument_list : argument_list COMMA ID EQUALS type_value'
subexpressions[0] = ArgumentAppend(subexpressions[3], subexpressions[5], subexpressions[1])

def p_type_value(self, subexpressions):
'type_value : variable'
subexpressions[0] = subexpressions[1]

def p_type_value_style(self, subexpressions):
'type_value : QUOTATION_MARK argl_style QUOTATION_MARK'
subexpressions[0] = subexpressions[2]

def p_argument_list_style_single(self, subexpressions):
'argl_style : ID COLON style_var SEMICOLON'
subexpressions[0] = StyleList(subexpressions[1], subexpressions[3])

def p_argument_list_style_append(self, subexpressions):
'argl_style : argl_style ID COLON style_var SEMICOLON'
subexpressions[0] = StyleAppend(subexpressions[2], subexpressions[4], subexpressions[1])

def p_argl_style_string(self, subexpressions):
'style_var : ID'
subexpressions[0] = ID(subexpressions[1])

def p_argl_style_num(self, subexpressions):
'style_var : NUMBER'
subexpressions[0] = Number(subexpressions[1])

def p_variable_number(self, subexpressions):
```

```
'variable : NUMBER'
subexpressions[0] = Number(subexpressions[1])

def p_variable_string(self, subexpressions):
'variable : STRING'
subexpressions[0] = String(subexpressions[1])

def p_variable_point(self, subexpressions):
'variable : LPAREN variable COMMA variable RPAREN'
subexpressions[0] = Point(subexpressions[2], subexpressions[4])

def p_variable_array(self, subexpressions):
'variable : LBRACKET variable_list RBRACKET'
subexpressions[0] = Array(subexpressions[2])

def p_variable_list_one(self, subexpressions):
'variable_list : variable'
subexpressions[0] = VariableList(subexpressions[1])

def p_variable_list_append(self, subexpressions):
'variable_list : variable_list COMMA variable'
subexpressions[0] = VariableAppend(subexpressions[1], subexpressions[3])

def find_column(self, t):
last_cr = self.code.rfind('\n', 0, t.lexpos)
if last_cr < 0: last_cr = 0
column = (t.lexpos - last_cr) + (1 if last_cr == 0 else 0)
return column

def p_error(self, t):

# when t is not defined (missing lexeme) we cant say where we are
if t is None:
print 'Parse error.'
raise self.ParseException()

line = t.lineno
column = self.find_column(t)
print 'Parse error at line %d, column %d:' % (line, column)
print self.code.split('\n')[line - 1]
print ' ' * (column - 1) + '^'
raise self.ParseException()

def parse(self, code):
self.code = code
return self.parser.parse(code, lexer=self.lexer.lexer)

def __init__(self, code):
self.availableFunctions = AvailableFunctions()
self.code = code
self.lexer = DibuLexer(code)
self.parser = yacc.yacc(module=self)

if __name__ == "__main__":

if len(argv) != 3:
print "Invalid arguments."
print "Use:"
print "  parser.py input.dibu output.svg"
exit()
```

```
filename = argv[1]
outfile  = argv[2]

input_file = open(filename, "r")
code = input_file.read()
input_file.close()

parser = DibuParser(code)

try:
expression = parser.parse(code)
except DibuParser.ParseException as exception:
print str(exception)
exit()
except DibuLexer.LexicalException as exception:
print str(exception)
exit()

try:
output = expression.evaluate()
except SemanticException as exception:
print str(exception)
exit()

pretty_xml = xmlParse(output).toprettyxml();

output_file = open(outfile, "w")
output_file.write(pretty_xml)
output_file.close()

print "SVG Generated!"
```

8.3 expressions.py

```
from dibu import *

class SemanticException(Exception):
    pass

class Expression(object):

def evaluate(self):
# Aca se implementa cada tipo de expresion.
raise NotImplementedError

class Start(Expression):

def __init__(self, expressions, f):
self.f = f
self.expressions = expressions

def evaluate(self):
tree_expressions = self.expressions.evaluate()
tree_expressions.reverse()

svg = SVG(tree_expressions, self.f)
result = svg.generateSVG();

return result

class ExpressionList(Expression):
```

```
def __init__(self, function_id, argument_list, next_expression,f):
self.f = f;
self.function_id = function_id;
self.argument_list = argument_list;
self.next_expression = next_expression;

def evaluate(self, size_defined = False):

function_id     = self.function_id
argument_list   = self.argument_list.evaluate()
next_expression = self.next_expression.evaluate()

# check if function exists
if not self.f.functionExists(function_id):
raise SemanticException("Non-existant function "+function_id+".")

# check if size was already defined
if function_id == "size":
if size_defined:
raise SemanticException("Size was already defined.")
else:
size_defined = True

# check if mandatory attributes are defined
attribute_types = [x[0] for x in argument_list]
mandatoryAttributes = [x[0] for x in self.f.getMandatoryAttributes(function_id)]
for attribute in mandatoryAttributes:
if attribute not in attribute_types:
raise SemanticException("Missing attribute "+attribute+" in "+function_id+".")

# check if all attributes are valid
attribute_types = [x[0] for x in argument_list]
mandatoryAttributes = [x[0] for x in self.f.getAllAttributes(function_id)]
for attribute in attribute_types:
if attribute not in mandatoryAttributes:
print function_id
print attribute
raise SemanticException("Attribute "+attribute+" is invalid.")

# check no attributes are defined twice
attribute_types = [x[0] for x in argument_list]
unique_attributes = set(attribute_types);
for attribute in attribute_types:
if attribute in unique_attributes:
unique_attributes.remove(attribute)
else:
raise SemanticException("Multiple definitions of attribute "+attribute+".")

# check if all attribute types are valid
possible_attributes = self.f.getAllAttributes(function_id)
for attribute_name, attribute_values in argument_list:
if attribute_name == "style":
style_attributes = self.f.getStyleAttributes()
for name, value in attribute_values:
if (name, value['type']) not in style_attributes:
raise SemanticException("Invalid style attribute type for "+name+".")
elif self.f.getAttributeType(function_id, attribute_name) == 'point':
for chord in attribute_values:
if chord != 'type' and attribute_values[chord]['type'] is not 'number':
raise SemanticException("Invalid attribute type for "+chord+" in point.")
```

```
elif self.f.getAttributeType(function_id, attribute_name) == 'array':
for point in attribute_values['value']:
if point['x']['type'] != 'number' or point['x']['type'] != 'number':
raise SemanticException("Array must be composed of valid points.")
else:
attribute = (attribute_name, attribute_values['type'])
if attribute not in possible_attributes:
raise SemanticException("Invalid attribute type for "+attribute[0]+".")

# semantic checks done
expression_list = self.next_expression.evaluate()
expression_list.append((function_id, argument_list))
return expression_list

class EmptyExpression(Expression):

def __init__(self):
pass

def evaluate(self):
return [];

class ArgumentList(Expression):

def __init__(self, function_id, variable):
self.function_id = function_id
self.variable    = variable

def evaluate(self):
function_id = self.function_id
variable    = self.variable.evaluate()

arg_list = []
arg_list.append((function_id, variable))
return arg_list

class ArgumentAppend(Expression):

def __init__(self, function_id, variable, argument_list):
self.function_id   = function_id
self.variable      = variable
self.argument_list = argument_list

def evaluate(self):
function_id = self.function_id
variable = self.variable.evaluate()

argument_list = self.argument_list.evaluate()

argument_list.append((function_id, variable))

return list(argument_list)

class StyleList(Expression):

def __init__(self, function_id, variable):
self.function_id = function_id
self.variable    = variable

def evaluate(self):
function_id = self.function_id
```

```
variable    = self.variable.evaluate()

arg_list = []
arg_list.append((function_id, variable))
return arg_list

class StyleAppend(Expression):

def __init__(self, function_id, variable, argument_list):
self.function_id = function_id
self.variable = variable
self.argument_list = argument_list

def evaluate(self):
function_id = self.function_id
argument_list = self.argument_list.evaluate()
variable = self.variable.evaluate()

argument_list.append((function_id, variable))

return list(argument_list)

class ID(Expression):

def __init__(self, value):
self.value = value;

def evaluate(self):
return {'value': self.value, 'type': 'string'}

class Number(Expression):

def __init__(self, value):
self.value = value;

def evaluate(self):
return {'value': self.value, 'type': 'number'}

class String(Expression):

def __init__(self, value):
self.value = value;

def evaluate(self):
return {'value': self.value, 'type': 'string'}

class Point(Expression):

def __init__(self, x, y):
self.x = x;
self.y = y;

def evaluate(self):
return {'x': self.x.evaluate(), 'y': self.y.evaluate(), 'type': 'point'}

class Array(Expression):

def __init__(self, variable_list):
self.variable_list = variable_list

def evaluate(self):
```

```
value = self.variable_list.evaluate();
return {'value': list(value), 'type': 'array'}

class VariableList(Expression):

def __init__(self, variable):
self.variable = variable;

def evaluate(self):
value = []
value.append(self.variable.evaluate())
return list(value)

class VariableAppend(Expression):

def __init__(self, variable_list, variable):
self.variable_list = variable_list
self.variable = variable;

def evaluate(self):
variable_list = self.variable_list.evaluate()
variable = self.variable.evaluate()

variable_list.append(variable)
return variable_list
```

8.4 dibu.py

```
from xml.etree import ElementTree as et

class AvailableFunctions(object):
"""Define available functions in our language."""

def __init__(self):

self.d = {}
self.optionalAttributes = {}
self.generalAttributes  = {}
self.styleAttributes    = {}

# these codes are used since the XML in the SVG
# functions does not necessarily match the function name
self.functionCodes  = {}
self.attributeCodes = {}

# define available functions
self.add('size');
self.addAttribute('size', 'height', 'number');
self.addAttribute('size', 'width',  'number');

self.add('rectangle', 'rect');
self.addAttribute('rectangle', 'upper_left', 'point');
self.addAttribute('rectangle', 'size',  'point');

self.add('line');
self.addAttribute('line', 'from', 'point');
self.addAttribute('line', 'to', 'point');

self.add('circle');
self.addAttribute('circle', 'center', 'point');
```

```
self.addAttribute('circle', 'radius', 'number', 'r');

self.add('ellipse');
self.addAttribute('ellipse', 'center', 'point');
self.addAttribute('ellipse', 'rx', 'number');
self.addAttribute('ellipse', 'ry', 'number');

self.add('polyline');
self.addAttribute('polyline', 'points', 'array');

self.add('polygon');
self.addAttribute('polygon', 'points', 'array');

self.add('text');
self.addAttribute('text', 't', 'string');
self.addAttribute('text', 'at', 'point');
# optional
self.addOptionalAttribute('text', 'font-familly', 'string');
self.addOptionalAttribute('text', 'font-size', 'string');

self.addGeneralAttribute('style', 'list')
self.addGeneralAttribute('fill', 'string')
self.addGeneralAttribute('stroke', 'string')
self.addGeneralAttribute('stroke-width', 'number')

self.addStyleAttribute('fill', 'string')
self.addStyleAttribute('stroke', 'string')
self.addStyleAttribute('stroke-width', 'number')

def print_functions(self):
print self.d;

def add(self, function_name, function_code = False):
if function_name not in self.d:
self.d[function_name] = {}
if function_name not in self.optionalAttributes:
self.optionalAttributes[function_name] = {}
if function_code:
self.functionCodes[function_name] = function_code;

def addAttribute(self, function_name, attribute_name, attribute_type, attribute_code = False):
if function_name not in self.d:
raise ValueError("You must first add the function "+function_name+".")
self.d[function_name][attribute_name] = attribute_type;
if attribute_code:
if function_name not in self.attributeCodes:
self.attributeCodes[function_name] = {}
self.attributeCodes[function_name][attribute_name] = attribute_code

def addOptionalAttribute(self, function_name, attribute_name, attribute_type):
self.optionalAttributes[function_name][attribute_name] = attribute_type;

def addGeneralAttribute(self, attribute_name, attribute_type):
self.generalAttributes[attribute_name] = attribute_type;

def addStyleAttribute(self, attribute_name, attribute_type):
self.styleAttributes[attribute_name] = attribute_type;

def setFunctionCode(self, function_name, function_code):
self.functionCodes[function_name] = function_code;
```

```
def getMandatoryAttributes(self, function_name):
attr_list = []
for attr, attr_type in self.d[function_name].items():
attr_list.append((attr, attr_type))
return attr_list;

def getAllAttributes(self, function_name):
attr_list = []
for attr, attr_type in self.d[function_name].items():
attr_list.append((attr, attr_type))
for attr, attr_type in self.optionalAttributes[function_name].items():
attr_list.append((attr, attr_type))
for attr, attr_type in self.generalAttributes.items():
attr_list.append((attr, attr_type))
return attr_list;

def getAttributeType(self, function_name, attribute_name):

if attribute_name in self.d[function_name]:
return self.d[function_name][attribute_name]

if attribute_name in self.optionalAttributes[function_name]:
return self.optionalAttributes[function_name][attribute_name]

if attribute_name in self.generalAttributes:
return self.generalAttributes[attribute_name]

print function_name
print attribute_name

raise ValueError("Non-existant attribute name.")

def getStyleAttributes(self):
attr_list = []
for attr, attr_type in self.styleAttributes.items():
attr_list.append((attr, attr_type))
return attr_list;

def getFunctionCode(self, function_name):
return self.functionCodes.get(function_name, function_name)

def getAttributeCode(self, function_name, attribute_name):
return self.attributeCodes.get(function_name, {}).get(attribute_name, attribute_name)

def functionExists(self, function_name):
return function_name in self.d;

def attributeExists(self, function_name, attribute_name):
return self.functionExists(function_name) and attribute_name in self.d[function_name];

def functionHasAttribute(self, function_name, attribute_name, attribute_type):
return self.functionExists(function_name) and (self.d[function_name][attribute_name] == attribute_type
or self.generalAttributes.get(attribute_name, 0) == attribute_type);

class SVG(object):

def __init__(self, expressions, f):
self.expressions = expressions;
self.f = f

def generateSVG(self):
```

```
doc = et.Element('svg', version='1.1', xmlns='http://www.w3.org/2000/svg')

# search for SVG size in expression tree
size_set = False
for function, argument_list in self.expressions:
if function == 'size':
for attribute, value in argument_list:
doc.attrib[attribute] = str(value['value']);
size_set = True
break


# sometimes size is not defined, get the size from a rectangle
if not size_set:
for function, argument_list in self.expressions:
if function == 'rectangle' and not size_set:
for attribute, value in argument_list:
if attribute == 'size':
doc.attrib['height'] = str(value['y']['value']);
doc.attrib['width']  = str(value['x']['value']);
size_set = True
break


# add the rest of the attributes
for function, argument_list in self.expressions:

if function == 'size': continue;

if function == 'text':
print argument_list
# [('t', {'type': 'string', 'value': 'I love Dibu'}),
#  ('at', {'y': {'type': 'number', 'value': 15}, 'x': {'type': 'number', 'value': 0}, 'type': 'point'})]
text = et.Element('text')

for attribute, value in argument_list:
if attribute == 'at':
text.attrib['x'] = str(value['x']['value'])
text.attrib['y'] = str(value['y']['value'])
elif attribute == 't':
text.text = value['value']
else:
print attribute
print value
text.attrib[self.f.getAttributeCode(function, attribute)] = str(value['value'])

doc.append(text)
else:
new_element = et.SubElement(doc, self.f.getFunctionCode(function));
for attribute, value in argument_list:

# print argument_list
# style attribute does not have a value with type but a list
if attribute == 'style':
# ('style', [('stroke', {'type': 'string', 'value': 'black'}),
# ('stroke-width', {'type': 'number', 'value': 3.0}),
# ('fill', {'type': 'string', 'value': 'none'})])]
string = []
for e in value:
string.append(e[0] + ': ' + str(e[1]['value']) + ';');
```

```
new_element.attrib['style'] = ' '.join(string)
elif value['type'] == 'point':
if attribute == 'size':
new_element.attrib['height'] = str(value['x']['value'])
new_element.attrib['width']  = str(value['y']['value'])
elif function == 'circle':
new_element.attrib['cx'] = str(value['x']['value'])
new_element.attrib['cy'] = str(value['y']['value'])
else:
new_element.attrib['x'] = str(value['x']['value'])
new_element.attrib['y'] = str(value['y']['value'])
elif value['type'] == 'array':
points = []
for point in value['value']:
points.append(str(point['x']['value'])+","+str(point['y']['value']))
new_element.attrib[self.f.getAttributeCode(function, attribute)] = ' '.join(points)
else:
new_element.attrib[self.f.getAttributeCode(function, attribute)] = str(value['value'])

string  = "<?xml version=\"1.0\" standalone=\"no\"?>\n";
string += et.tostring(doc)

return string
```