Teoria de Lenguajes

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

Trabajo Práctico

Dibu: Graficos vectoriales para ninos

Integrante	LU	Correo electrónico
Julián Bayardo	850/13	julian@bayardo.com.ar
Christian Cuneo	755/13	chriscuneo93@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

Se nos presenta un lenguaje con una estructura especifica y se nos pide que generemos un analizador léxico y un analizador sintáctico para poder decidir la pertenencia de una cadena a este lenguaje, y si es así, obtener la estructura de esta cadena. El lexer nos va a permitir reconocer los elementos del lenguaje que se encuentran en la cadena. Y el parser (analizador sintáctico) nos va a indicar si lo que fue escrito con esos elementos tiene sentido (cumple la estructura del lenguaje).

En nuestro caso el lenguaje describe una forma de construir un conjunto figuras geométricas que van a componer un gráfico. Para lograr esto el lenguaje consiste en una serie de instrucciones que van a definir distintos tipos de figuras y el gráfico en su totalidad.

Luego buscamos también lograr transformar la estructura de la cadena parseada en una cadena valida del lenguaje SVG (que funciona sobre xml)

Los dos son problemas separados, pero básicamente consiste en obtener la estructura de la cadena de entrada (si esta es valida) y transformarla elemento por elemento en una cadena valida SVG.

2. Gramática

La gramática que generamos que describe todas las cadenas de este lenguaje es la siguiente:

- $S \rightarrow statement$
- $statement \rightarrow expression$
- $statement \rightarrow statement\, expression$
- $expression \rightarrow IDENTIFIER\, key\_value\_list$
- $key\_value\_list \rightarrow key\_value\_entry\, COMMA\, key\_value\_list$
- $key\_value\_list \rightarrow key\_value\_entry$
- $key\_value\_entry \rightarrow KEY\ EQUALS\, value$
- $value \rightarrow STRING$
- $value \rightarrow NUMBER$
- $value \rightarrow LPAREN\, value\, COMMA\, value\, RPAREN$
- $value \rightarrow LBRACKET\, array\, RBRACKET$
- $array \rightarrow value$
- $array \rightarrow value\, COMMA\, array$

Donde los elementos en mayúscula son símbolos terminales que matchean con las siguientes expresiones regulares:

```
IDENTIFIER = (size|rectangle|line|circle|ellipse|polyline|polygon|
    ↪ text)
STRING = \"([^\\\n]|(\\.))*?(?<!\\)"
KEY = \w[\w\d_-]*
COMMA = ,
EQUALS = =
LBRACKET = \[
RBRACKET = \]
LPAREN = \(
RPAREN = \)
NUMBER = \d+
```

Podemos observar que la gramática definida no es LL(1) por las reglas de statement, key_value_list y array; además, no es LR(0) por tener varios conflictos shift reduce: uno ocasionado por la regla $S \rightarrow statement$, otro ocasionado por la regla $key\_value\_list \rightarrow key\_value\_entry$, y otro por la regla $array \rightarrow value$. Puede verse (o bien haciendo el autómata a mano, o bien a través de PLY, o bien utilizando una herramienta como el Grammophone), que la gramática es SLR(1) (y por ende también LALR y LR(1)). De hecho, nuestra solución utiliza el método SLR de PLY.

3. Solución

La solución se realizo utilizando el esqueleto provisto por la cátedra, quiere decir usando *python* y la librería *ply* de ese lenguaje. Para correr el parser se utilizo el *Notebook* de *Jupyter* dado por la cátedra pero levemente modificado (se encuentra en el código fuente)

El código se encuentra también en el informe pero adjuntado al final del mismo para una mayor comodidad de lectura.

Para la implementación utilizamos como referencia el código de los distintos lexers y parsers provistos por la cátedra. No nos surgieron problemas reales al implementar la solución mas que entender como funcionaba la librería utilizada. Tuvimos problemas simplemente al querer obtener la fila actual al estar parseando una subexpresión, mas que nada para levantar errores que sean auto explicativos y concisos, pero otra vez, fue mas un problema de entender como funcionaba la librería *ply*.

También decidimos levantar errores semánticos detallados cuando la cadena no cumple la semántica del lenguaje (osea, que quizás lexicografica y sintácticamente es correcta pero no tiene sentido).

4. Como usar nuestro parser

Como indicamos previamente la forma mas fácil de utilizarlo es a través del *Notebook* de *Jupyter*. Utilizamos *Python* 3,5 para implementar y correr el código. Luego los requerimientos están especificados en el archivo *requirements.txt* por lo tanto se pueden instalar fácilmente con *pip*

En el *notebook dibu.ipynb* dejamos se encuentran ejemplos de como correr la herramienta.

Tambien se puede correr desde la carpeta principal del source de la forma:
$python3 -m\, dibu.parser\, '\#\#\#CADENA\, A\, PARSEAR\#\#\#'$

5. Casos de prueba

5.1. Círculos superpuestos

Input:

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

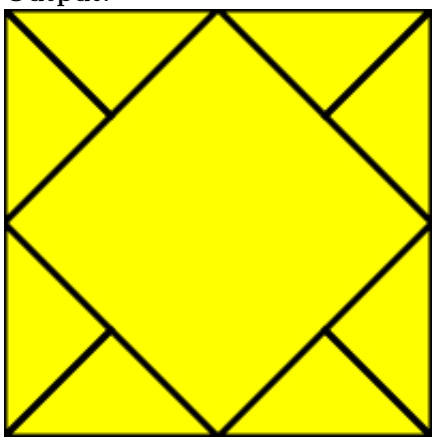
Output:

5.2. Cuadrado con polígonos internos

Input:

```
size height=200, width=200
rectangle upper_left=(0,0), width=200, height=200,fill="yellow"
polygon points=[(0,0), (50, 50), (0, 100)], stroke="black", stroke-
    ↪ width=3, fill="none"
polygon points=[(0,0), (50, 50), (100, 0)], stroke="black", stroke-
    ↪ width=3, fill="none"
polygon points=[(0, 100), (50, 150), (0, 200)],stroke="black", stroke-
    ↪ width=3, fill="none"
polygon points=[(0, 200), (50, 150), (100, 200)],stroke="black", stroke
    ↪ -width=3, fill="none"
polygon points=[(100, 200), (150, 150), (200,200)], stroke="black",
    ↪ stroke-width=3, fill="none"
polygon points=[(200, 200), (150, 150), (200,100)], stroke="black",
    ↪ stroke-width=3, fill="none"
polygon points=[(200, 100), (150, 50), (200, 0)],stroke="black", stroke
    ↪ -width=3, fill="none"
polygon points=[(200, 0), (150, 50), (100, 0)],stroke="black", stroke-
    ↪ width=3, fill="none"
```

Output:



5.3. Error semántico (size definido 2 veces)

Input:

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

Output:

```
SemanticException: Line 3: Size defined twice.
```

5.4. Cuadrado con puntas redondeadas y poco opaco

Input:

```
size width=400, height=180
rectangle upper_left=(50,20), rx=20, ry=20, width=150, height=150,
    ↪ style="fill:red;stroke:black;stroke-width:5;opacity:0.5"
```

Output:

5.5. Cuadrado con circulos adentro y texto abajo

Input:

```
size width=400, height=300
rectangle upper_left=(50,20), rx=20, ry=20, width=150, height=150,
    ↪ style="fill:red;stroke:black;stroke-width:5;opacity:0.5"
circle center=(125, 95), radius=50, fill="red"
circle center=(125, 95), radius=25, fill="black"
text at=(10, 200), t="Metiste un cutucuchillo y te pode queda
    ↪ lectrificada loca
```

Output:

5.6. Error semántico parámetro no existe para rectangulo

Input:

```
size width=400, height=180
rectangle mama=(50,20)
```

Output:

```
SemanticException: Line 1: Parameters ['upper_left'] need to be
    ↪ defined for rectangle
```

5.7. Error sintáctico identificador token no existe

Input:

```
size width=400, height=180
rectangsdale upper_left=(50,20)
```

Output:

```
Exception: [Syntax error]
type:KEY
value:rectangsdale
line:2
position:31
```

6. Conclusiones

Los resultados fueron correctos. Nos tomamos la libertad de agregar el parámetro opcional rx y ry al *rectangle* y *style* a todos para ver resultados mas interesantes.

Encontramos interesante implementar el lexer y parser ya que nos deja ver de forma muy clara como se utiliza una gramática para ver si una cadena pertenece a un lenguaje.

7. Código

7.1. Parser

```
import ply.yacc as yacc
import ply.lex as lex
from .lexer_rules import *
from .parser_rules import *


def parse(text, debug=False):
    lexer = lex.lex(debug=debug)
    parser = yacc.yacc(method='SLR', debug=debug)
    document = parser.parse(text, lexer)

    canvas = '<svg xmlns="http://www.w3.org/2000/svg" version="1.1">'
    shapes = []

    for key, value in reversed(document):
        if key == 'size':
            canvas = '<svg xmlns="http://www.w3.org/2000/svg" version
                ↪ ="1.1" width="{}" height="{}">'.format(value['width'
                ↪ ], value['height'])
            continue
        if key == 'rectangle':
            element = '<rect x="{}" y="{}" height="{}" width="{}" '.
                ↪ format(
                value['upper_left'][0],
                value['upper_left'][1],
                value['height'],
                value['width']
            )
        elif key == 'line':
            element = '<line x1="{}" y1="{}" x2="{}" y2="{}" '.format(
                value['from'][0],
                value['from'][1],
                value['to'][0],
                value['to'][1]
            )
        elif key == 'circle':
            element = '<circle cx="{}" cy="{}" r="{}" '.format(
                value['center'][0],
                value['center'][1],
                value['radius']
            )
        elif key == 'ellipse':
            element = '<ellipse cx="{}" cy="{}" rx="{}" ry="{}" '.format
                ↪ (
                value['center'][0],
                value['center'][1],
                value['rx'],
                value['ry']
            )
        elif key == 'polyline':
            element = '<polyline points="{}" '.format(
                " ".join(["{},{}".format(point[0], point[1]) for point
```

```
                    ↪ in value['points']])
            )
        elif key == 'polygon':
            element = '<polygon points="{}"'.format(
                " ".join(["{},{}".format(point[0], point[1]) for point
                    ↪ in value['points']])
            )
        elif key == 'text':
            element = '<text x="{}" y="{}"'.format(
                value['at'][0],
                value['at'][1]
            )
            if 'font-family' in value:
                element += ' font-family="{}"'.format(value['font-
                    ↪ family'])
            if 'font-size' in value:
                element += ' font-size="{}"'.format(value['font-size'])
        for par in common:
            if par in value:
                element += ' {}="{}"'.format(
                    par,
                    value[par]
                )
        if key == 'text':
            element += '>{}</text>'.format(value['t'])
        else:
            element += '/>'
        shapes.append(element)
    return "{}{}</svg>".format(canvas, "".join(shapes))
```

7.2. Reglas del lexer

```
tokens = [
    'IDENTIFIER',
    'STRING',
    'NUMBER',
    'KEY',
    'COMMA',
    'EQUALS',
    'LBRACKET',
    'RBRACKET',
    'LPAREN',
    'RPAREN'
]


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


t_IDENTIFIER = r'(size|rectangle|line|circle|ellipse|polyline|polygon|
```

```
    ↪ text)'
t_STRING = r'\"([^\\\n]|(\\.))*?(?<!\\)"'
t_KEY = r'\w[\w\d_-]*'
t_COMMA = r","
t_EQUALS = r"="
t_LBRACKET = r"\["
t_RBRACKET = r"\]"
t_LPAREN = r"\("
t_RPAREN = r"\)"


def t_NUMBER(t):
    r'\d+'
    try:
        t.value = int(t.value)
    except ValueError:
        print("Integer value too large", t.value)
        t.value = 0
    return t


def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

t_ignore = ' \t'
```

7.3. Reglas del parser

```
required = {
    'size': {
        'height': int,
        'width': int
    },
    'rectangle': {
        'upper_left': (int, int),
        'height': int,
        'width': int
    },
    'line': {
        'from': (int, int),
        'to': (int, int)
    },
    'circle': {
        'center': (int, int),
        'radius': int
    },
    'ellipse': {
        'center': (int, int),
        'rx': int,
        'ry': int
    },
    'polyline': {
        'points': [(int, int)]
    },
    'polygon': {
```

```
        'points': [(int, int)]
    },
    'text': {
        't': str,
        'at': (int, int)
    }
}

optional = {
    'text': {
        'font-family': str,
        'font-size': str
    }
}

common = {
    'fill': str,
    'stroke': str,
    'stroke-width': int
}


class SemanticException(Exception):
    pass


def type_assert(a, b):
    if type(b) == tuple:
        if type(a) is not tuple:
            return False

        for x, y in zip(a, b):
            if not isinstance(x, y):
                return False

        return True
    elif type(b) == list:
        if type(a) is not list:
            return False

        b = b[0]
        for entry in a:
            if not type_assert(entry, b):
                return False

        return True
    else:
        return isinstance(a, b)


def p_statement(subexpressions):
    '''statement : expression
                 | statement expression'''

    if len(subexpressions) == 3:
        subexpressions[0] = [subexpressions[2]]
```

```
        subexpressions[0] += subexpressions[1]

        if subexpressions[2][0] == 'size':
            for entry in subexpressions[1]:
                if entry[0] == 'size':
                    raise SemanticException("Line {}: Size defined
                        ↪ twice.".format(subexpressions.lexer.lineno))
    else:
        subexpressions[0] = [subexpressions[1]]


def p_expression(subexpressions):
    'expression : IDENTIFIER key_value_list'
    subexpressions[0] = (subexpressions[1], subexpressions[2])

    (identifier, arguments) = subexpressions[0]

    unmet_requirements = []
    for key in required[identifier]:
        if key not in arguments:
            unmet_requirements.append(key)

    if len(unmet_requirements) > 0:
        raise SemanticException(
            "Line {}: Parameters {} need to be defined for {}".format(
                ↪ subexpressions.lexer.lineno - 1, unmet_requirements,
                ↪  identifier))

    # Check types and key validity
    for key in arguments:
        if key in common:
            if type_assert(arguments[key], common[key]):
                return
            raise SemanticException(
                "Line {}: Type of parameter {} for {} is {}, not {}".
                    ↪ format( subexpressions.lexer.lineno - 1, key,
                    ↪ identifier, common[key], arguments[key]))
        elif key in required[identifier]:
            if type_assert(arguments[key], required[identifier][key]):
                return
            raise SemanticException(
                "Line {}: Type of parameter {} for {} is {}, not {}".
                    ↪ format( subexpressions.lexer.lineno - 1, key,
                    ↪ identifier, required[key], arguments[key]))
        elif identifier in optional and key in optional[identifier]:
            if type_assert(arguments[key], optional[identifier][key]):
                return
            raise SemanticException(
                "Line {}: Type of parameter {} for {} is {}, not {}".
                    ↪ format( subexpressions.lexer.lineno - 1, key,
                    ↪ identifier, optional[identifier][key], arguments
                    ↪ [key]))
        else:
            raise SemanticException(
                "Line {}: Unknown parameter {} for {}.".format(
                    ↪ subexpressions.lexer.lineno - 1, key, identifier
```

```
                    ↪ ))


def p_key_value_list(subexpressions):
    '''key_value_list : key_value_entry COMMA key_value_list
                      | key_value_entry
    '''

    kv = subexpressions[1]

    if len(subexpressions) == 4:
        for key in subexpressions.slice[3].value:
            if key in kv:
                raise SemanticException('Double definition for key {}'.
                    ↪ format(key))
        kv.update(subexpressions.slice[3].value)

    subexpressions[0] = kv


def p_key_value_entry(subexpressions):
    'key_value_entry : KEY EQUALS value'
    subexpressions[0] = {
        subexpressions[1]: subexpressions[3]
    }


def p_value(subexpressions):
    '''value : STRING
             | NUMBER
             | LPAREN value COMMA value RPAREN
             | LBRACKET array RBRACKET'''

    if subexpressions.slice[1].type == 'STRING':
        subexpressions[0] = str(subexpressions.slice[1].value)[1:-1]
    elif subexpressions.slice[1].type == 'NUMBER':
        subexpressions[0] = int(subexpressions.slice[1].value)
    elif subexpressions.slice[1].type == 'LPAREN':
        subexpressions[0] = (subexpressions[2], subexpressions[4])
    elif subexpressions.slice[1].type == 'LBRACKET':
        subexpressions[0] = subexpressions[2]


def p_array(subexpressions):
    '''array : value
             | value COMMA array'''

    subexpressions[0] = [subexpressions[1]]

    if len(subexpressions) == 4:
        subexpressions[0] += subexpressions[3]


def p_error(token):
    message = "[Syntax error]"
    if token is not None:
```

```
        message += "\ntype:" + token.type
        message += "\nvalue:" + str(token.value)
        message += "\nline:" + str(token.lineno)
        message += "\nposition:" + str(token.lexpos)
    raise Exception(message)
```