



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico I

Grupo: Parsers para todos

Teoría de Lenguajes  
Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Federico De Rocco	408/13	fede.183@hotmail.com
José Massigoge	954/12	jmmassigoge@gmail.com
Leandro Anacondio	487/07	leandro.anacondio@gmail.com



Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

# Índice

1. Introducción	3
2. Gramática	4
3. Lexer	6
4. Parser	7
5. Requerimientos de Software	9
6. Test	10
7. Conclusiones	11

## 1. Introducción

En el presente Trabajo Práctico diseñamos una gramática para el lenguaje *Dibu* e implementamos sus analizador léxico y su analizador sintáctico y semántico (parser). Utilizando estas herramientas creamos un programa que permite traducir de *Dibu* a *SVG* y compilar dicha traducción, en caso de que sea válida. En los siguientes items se describirán la gramática para el lenguaje, el lexer y el parser.

## 2. Gramática

En la siguiente sección vamos a mostrar y describir la gramática construida para el lenguaje *Dibu*, llamo a esta gramática  $G$ :

$P \rightarrow S \text{ newline } P \mid \lambda$   
 $S \rightarrow \text{id PARAMS}$   
 $\text{PARAMS} \rightarrow \text{id} = V, \text{PARAMS} \mid \text{id} = V$   
 $V \rightarrow \text{num} \mid \text{string} \mid (\text{num}, \text{num}) \mid [\text{ARRAY}]$   
 $\text{ARRAY} \rightarrow [\text{num}, \text{num}], \text{ARRAY} \mid [\text{num}, \text{num}]$

$G = \{\{P, S, \text{PARAMS}, V, \text{ARRAY}\}, \{\text{num}, \text{string}, (, , ), [, ], =, \text{id}, \text{newline}\}, \text{Descripto por la gramática, } P\}$

Como se describe en el enunciado, el lenguaje *Dibu* es una serie de instrucciones de la forma:

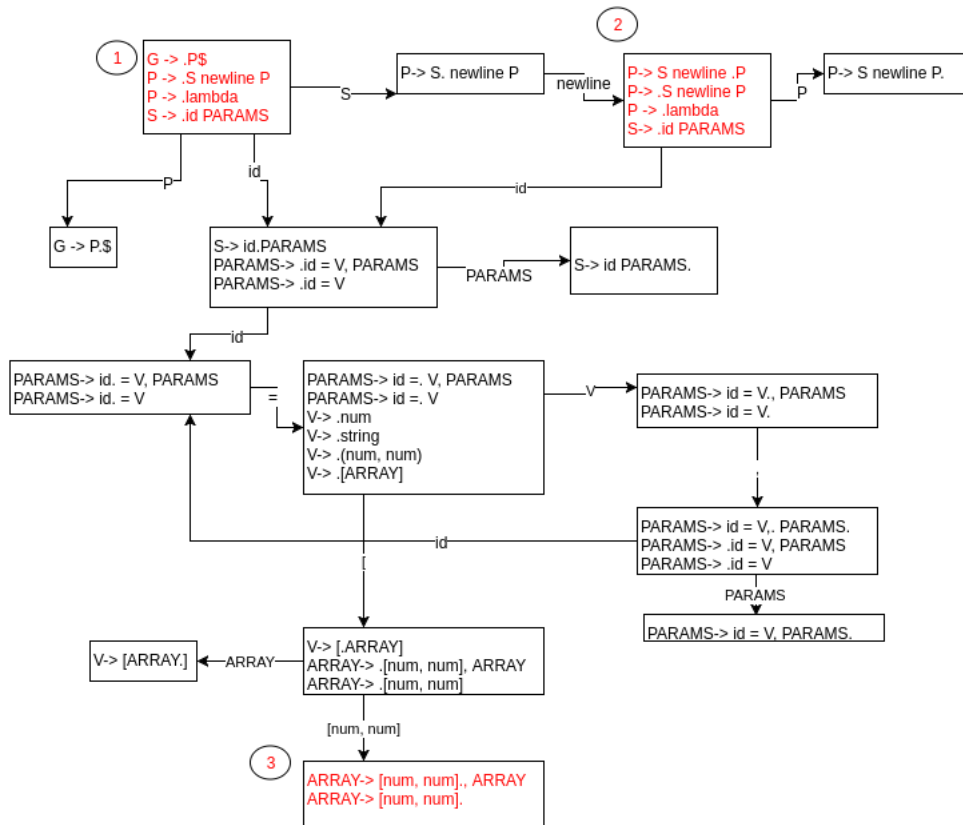
IDENTIFICADOR PARAM1=V1, PARAM2=V2, ..., PARAMN=VN

Nuestra gramática consiste en cinco producciones. Comenzando con  $\text{ARRAY}$  la cual describe la secuencia de valores numericos separados por una coma, la cual se utilizara en  $V$  para describir arreglos. La producción  $V$  que contiene las combinaciones de terminales que se pueden esperar como valores de los parámetros ( $\text{num}$ ,  $\text{string}$ ,  $\text{point}$ ,  $\text{array}$ ).  $\text{PARAMS}$  posee la serie de compuesta por: parámetro = valor.  $S$  describe una instrucción, con su nombre y parámetros. Y finalmente  $P$ , describe la serie compuesta de las instrucciones descritas en el no-terminal  $S$ . Con estas producciones se puede ver con estos que nuestra gramática describe precisamente la serie de instrucciones de *Dibu*. Se debe aclarar que esta gramática no describe todas las restricciones de *Dibu*, como por ejemplo que solo aparezca la instrucción `size` una vez. Estas custriones se tratarán en el lexer y el parser.

Por otro lado tenemos el tipo de la gramática, para exponer esta información no realizaremos una demostración rigurosa aunque si analizaremos la misma descartando en primera instancia que la gramática sea ambigua. Posteriormente reafirmaremos en que tipos de gramáticas no se cuenta y el por qué.

Esta gramática no es  $\text{LL}(1)$  porque las producciones de  $\text{PARAMS}$  no cumplen con el requerimiento de estas gramáticas ( $\text{SD}(\text{PARAMS} \rightarrow \text{id} = V \text{ PARAMS } P) \cup \text{SD}(\text{PARAMS} \rightarrow \text{id} = V)$ ).

La gramática tampoco es  $\text{LR}(0)$ . A continuación se mostrara el autómata correspondiente a la gramática, marcando los conflictos. Solamente se consideraron los estados del autómata que fueran interesantes por sus conflictos.



La gramática es SLR ya que conservando el mismo autómata vemos que los siguientes de P y ARRAY que son los implicados en las nodos que generan los conflictos son:

Siguientes(P) = { \$ }  
 Siguietes(ARRAY) = { ] }

Con lo cual los nodos 1 y 2 que presentaban conflictos por el lambda quedan resueltos debido a que solamente se reducirá por \$ y no por id. Por último el nodo conflictivo 3 solo reducirá con ] por lo que se eliminan todos los conflictos shift/reduce.

### 3. Lexer

En esta sección vamos a explicar el analizador léxico implementado para la gramática descrita anteriormente. Al aplicar un lexer sobre una cadena de caracteres este devolverá verdadero si la misma no contiene errores de carácter léxico como por ejemplo caracteres que el lenguaje no reconoce. Para llevar a cabo este analizador contamos con una serie de tokens de los cuales se da una descripción de sus posibles contenidos. Estos son:

**NUMBER:** Son todos los números compuestos por los caracteres numéricos del 0 al 9 y el punto. El lexer identifica el tipo del valor numérico siendo estos enteros y de punto flotante de acuerdo a si el número posee o no el punto. Esto se hace para el posterior control de errores.

**STRING:** Cadenas de caracteres entre comillas compuestas por las letras de la a a la z, incluidas mayúsculas, los números del 0 al 9 y los símbolos +, - y \*.

**ID:** Igual que **STRING** pero sin las comillas.

**LPAREN:** El carácter [.

**RPAREN:** El carácter ].

**LBRACKET:** El carácter (.

**RBRACKET:** El carácter ).

**COMMA:** El carácter ,.

**EQUALS:** El carácter =.

**NEWLINE:** El salto de línea. Este tiene la particularidad de que aumenta en una línea el largo del texto, esto se utiliza para detectar la posición de los errores.

**IGNORE:** El carácter vacío.

**ERROR:** Este aparece cuando se encuentra un token desconocido. En consecuencia se guarda la línea, valor, posición y tipo de valor para el error. Con este último cumplimos que si la cadena es inválida se detectara el error y se informará de él.

Como se puede ver con esta serie de tokens y sus respectivas implementaciones podemos describir todos los posibles no-terminales necesarios para nuestra gramática. Quedan todavía restricciones como por ejemplo que **ID** solamente puede tener como valores a los identificadores de las instrucciones.

## 4. Parser

En esta sección vamos a explicar el analizador sintáctico o parser implementado. El análisis sintáctico convertirá el texto de entrada en el árbol de derivación pertinente. A partir de ello hacemos el análisis de las sentencias, utilizando estructuras auxiliares para cumplir posteriormente con el requerimiento de la generación del texto de salida (en formato svg). La idea general fue definir una función que represente cada una de las producciones de nuestra gramática, y dentro de cada una ir manipulando la información para luego generar el output requerido. Como atributos utilizamos `lineno` y `lexpos` para detallar el número de línea y posición respectivamente, mientras que el atributo `value` nos devuelve el valor. Como estructuras auxiliares utilizamos un diccionario y una lista. El primero representa los parámetros obligatorios de cada una de las figuras, lo utilizamos dentro de la producción `STATE` para chequear que todos los parámetros obligatorios están dentro de la cadena de entrada para cada figura en particular. El segundo es simplemente una lista donde iremos acumulando las distintas figuras que se fueron generando, teniendo en cuenta que nuestro parser es bottom-up, las figuras se irán generando a partir de las hojas y una vez que se llegue a la producción inicial, tendremos la lista llena de las figuras que debemos imprimir. Agregamos una función que hiciera las veces de producción inicial solo a fines prácticos, para poder generar el lienzo final, e ir agregando las figuras que posteriormente dibujaremos. A continuación haremos un análisis de cada una de esas funciones para explicar cuál es su rol.

**P.START:** genera el lienzo llamando a la función `Scene` (definida en la clase `helper.py`) y luego le agrega las figuras de la lista al mismo. También chequea que no se haya llamado a la función `size` mas de una vez.

**P.PROGRAM\_EMPTY:** representa al programa que se genera a partir de la producción  $P \mid \lambda$

**P.PROGRAM\_NONEMPTY:** representa al programa que se genera a partir de la producción  $P \rightarrow S \text{ newline } P$

**P.STATE:** Es la función más compleja del parser. Primero chequea que el token sea `size`, si lo es, se revisa que tenga el `height` y `width` que son los parámetros requeridos. Si el token no es `size` estamos ante la producción para generar una figura, por ello, se guarda los parámetros recolectados por los nodos hijos en una variable, e inicializa la figura como objeto (correspondiente al nombre de la misma, dentro de una función auxiliar). Si la generación no lanza una excepción, va completando los atributos requeridos por esa figura y la agrega a la lista de resultados. Si no se genera bien la figura o si los parámetros son incorrectos se lanza la excepción correspondiente.

**P.PARAMS\_NONRECURSIVE:** Representa la asignación de un valor a un parámetro sin recursión, es decir es el último de la lista o es único. Dentro de esta función se agrega el número de línea, posición y valor del parámetro al diccionario de parámetros.

**P.PARAMS\_RECURSIVE:** Representa la seguidilla de parámetros separados por coma. Chequea que no haya repetidos y los va agregando a la lista de parámetros.

**P.VALOR\_NUMBER:** Realiza la asignación de un valor a un parámetro de tipo numérico.

**P.VALOR\_STRING:** Realiza la asignación de un valor a un parámetro de tipo string.

**P.VALOR\_POINT:** Realiza la asignación de un valor a un parámetro de tipo punto, es decir dos números separados por coma.

**P.VALOR\_ARRAY:** Realiza la asignación de un valor a un parámetro de tipo arreglo.

**P.ARRAY\_ELEMENT:** Representa el array con un único elemento, o el elemento final de un arreglo.

**P.ARRAY\_RECURSIVE:** Genera un nuevo elemento en un array y lo appendea a los elementos que siguen en la producción.

**P.ERROR:** Define los errores sintácticos que ocurrieron. A partir del token genera el mensaje correspondiente.

Nuestra implementación utiliza *Scene*, una clase pensada para construir una cadena de SVG a partir de los datos de los parámetros que tomamos de la cadena de *Dibu*. Después de declarar nuestra *Scene* se puede agregar líneas de figuras utilizando las funciones del mismo nombre que la figura y los parámetros. Al final, la cadena de *Scene* es una cadena SVG la cual podemos utilizar para generar la imagen pedida.

Nuestro parser actúa como una traducción dirigida por sintaxis que se encargara de analizar la cadena hasta encontrar algún error, como por ejemplo dos *size* en una cadena. Para esto se recurrirá a los siguientes atributos:

**lineno:** Es un numero entero que indica la linea actual. Lo poseen *num*, *string*, y todos los no-terminales.  
**lexpos:** Es un numero entero que indica la posición en la linea. Lo poseen *num*, *string*, y todos los no-terminales.

**value:** Indica el valor según el contexto en que se encuentra. Lo poseen *num*, *string* y todos los no-terminales excepto por *PARAMS*.

**parámetros:** Es un diccionario que asocia a los nombres de los parámetros con su valor en una linea. Lo tiene *PARAMS*.

Además de estos atributos, tenemos dos diccionarios. El primero es *diccSize* que usamos para guardarnos los *size's* que aparecen en la cadena, en el caso de que no exista ninguno entonces se usará el *Scene* sin especificar un tamaño. Si hay mas de uno entonces estamos ante un error semántico que debemos avisar. El segundo de estos diccionarios es el de nombre *idDicObligatory* que sirve para poder guardar de forma estática los parámetros obligatorios de cada figura. Esto sirve para comprobar si en nuestra cadena todas las figuras tienen al menos todos sus parámetros obligatorios.

Por último tenemos a *listaFiguras* la cual guarda las figuras construidas para posteriormente agregarse al *Scene*.

Utilizando los atributos y las estructuras adicionales cubrir todos los casos en el lenguaje *Dibu* tales como la repetición de *size*, que si o si cada figura tenga sus parámetros obligatorios una vez por cada uno y sin orden, lo mismo para los opcionales pero sin que tengan que estar todos y además el poder construir la cadena SVG usando *Scene*.

El programa en si toma un archivo de texto que contiene una cadena del lenguaje *Dibu* y el nombre que va a tener la imagen resultante del análisis y traducción del primer parámetro. Tenemos un *try-catch*, el cual usaremos para poder capturar los errores propios de una cadena de entrada invalida y posteriormente mostrarlos por pantalla dando la posición, linea y el tipo de error (Sintáctico, Semántico y de Tokens). Una vez generada la cadena traducida de SVG mediante el parser, se la ejecutara generando la imagen y guardándola con el nombre pasado como parámetro.



## 5. Requerimientos de Software

## 6. Test

## **7. Conclusiones**