DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico II

Teoría de las Comunicaciones

Primer 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

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 newline P | $\lambda$
S $\rightarrow$ id PARAMS
PARAMS $\rightarrow$ id = V PARAMS P | id = V
V $\rightarrow$ num | string | (num, num) | [ARRAY]
ARRAY $\rightarrow$ [num, num], ARRAY | [num, num]

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

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

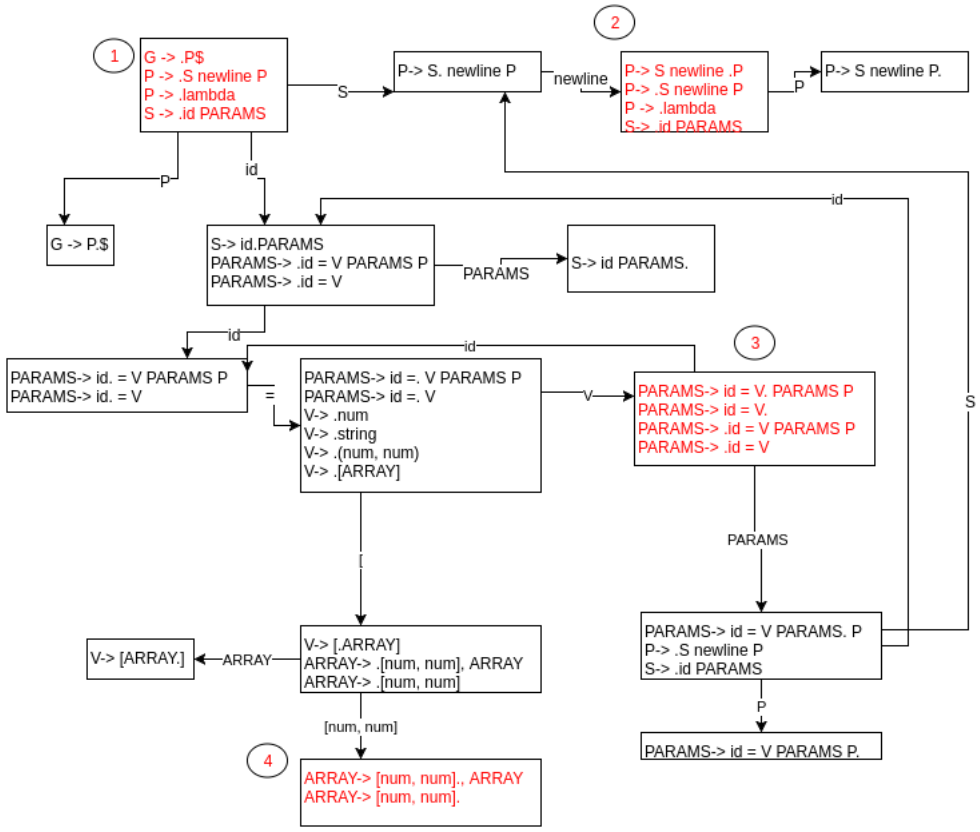
IDENTIFICADOR PARAM1=V1, PARAM2=V2, ..., PARAMN=VN

Nuestra gramática consiste en cinco producciones. Comenzando con 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 (num, string, point, array). 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 descriptas 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 cuestriones 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 LL(1) porque las producciones de PARAMS no cumplen con el requerimiento de estas gramáticas(SD(PARAMS $\rightarrow$ id = V PARAMS P) $\cup$ SD(PARAMS $\rightarrow$ id = V)).

La gramática tampoco es LR(0). A continuación se mostrara el autómata correspondiente a la gramática, marcando los conflictos.




No es SLR ya que los siguientes de PARAMS son {newline, id} por lo tanto no se resuelve el conflicto ya que hay un shift/reduce provocado por el terminal id aunque si resuelve los demás conflictos.

Finalmente nuestra gramática es LR(1), observemos que el primer conflicto quedaría resuelto ya que solo haría reduce en $, en el caso del segundo se resolvería de la misma forma. En el caso del tercero, en una producción anterior arrastraría el token newline y terminaría reduciendo por este, eliminando así el conflicto provocado por id. Por último tenemos el cuarto similar al anterior en las producciones anteriores se los asociaría con el token ] el cual eliminaría el conflicto con la coma.

# 3. Lexer

En esta sección vamos a explicar el analizador léxico implementado para la gramática descripta anteriormente. Al aplicar un lexer sobre una cadena de caracteres este devolverá verdadero su 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 numero 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 linea. Este tiene la particularidad de que aumenta en una linea 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 linea, 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, 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 cad una de las figuras, lo utilizamos dentro de la produccion 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 iran 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 | $\lambda$

P_PROGRAM_NONEMPTY: representa al programa que se genera a partir de la producción P $\rightarrow$ S 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 linea, 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.