DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Parser para gramática PEGS

27 de noviembre de 2014 — Teoría de Lenguajes de Programación

Grupo 10

Integrante	LU	Correo electrónico
Garbi, Sebastían	179/05	garbyseba@gmail.com
Sarries, Ana	144/02	anasarries@yahoo.com.ar

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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) 4576-3300
http://www.exactas.uba.ar

1. Introducción

Para la realización de este trabajo contamos con las reglas de un lenguaje simple con el cual se pueden generar composiciones visuales complejas con muy pocas lineas de definición. Este trabajo práctico consiste en generar un analizador léxico-sintáctico que pueda interpretar archivos escritos con ese lenguaje y luego darle semántica a lo interpretado generando la composición visual resultante. Para este cometido utilizaremos PLY[1] de Python, y los conocimientos aprendidos en la materia para configurar correctamente dicha herramienta.

2. Gramática

Para interpretar el lenguaje se generó la gramática $G = \langle V_n, V_t, P, Programa \rangle$ donde:

- V_n es: {Elemento, Elementoand, Elementobase, Factor, Main, Masreglas, Numero, Prim, Programa, Reglas, Termino, Trans, Unaregla}
- V_t es:{$, &, (,), *, +, -, ., /, :, <, =, >, BALL, BOX, CB, CG, CR, D, NADA, NUM, REGLA, RX, RY, RZ, S, SX, SY, SZ, TX, TY, TZ, [,], ^,| }
- Producciones P:
  $Programa \rightarrow Reglas\ Main\ Masreglas$
  $Reglas \rightarrow \lambda$
  $Reglas \rightarrow Reglas\ Unaregla$
  $Main \rightarrow \$\ =\ Elemento$
  $Main \rightarrow \$.\ =\ Elemento$
  $Masreglas \rightarrow \lambda$
  $Masreglas \rightarrow Masreglas\ Unaregla$
  $Masreglas \rightarrow Masreglas\ Main$
  $Unaregla \rightarrow \mathbf{REGLA}\ =\ Elemento$
  $Unaregla \rightarrow \mathbf{REGLA.}\ =\ Elemento$
  $Elemento \rightarrow Elemento \mid Elementoand$
  $Elemento \rightarrow Elementoand$
  $Elementoand \rightarrow Elementoand\ \&\ Elementobase$
  $Elementoand \rightarrow Elementobase$
  $Elementobase \rightarrow Prim$
  $Elementobase \rightarrow Elementobase\ :\ Trans$
  $Elementobase \rightarrow [\ Elemento\ ]$
  $Elementobase \rightarrow Elementobase\ \hat{}\ Numero$
  $Elementobase \rightarrow <\ Elemento\ >$
  $Elementobase \rightarrow \mathbf{REGLA}$
  $Elementobase \rightarrow \$$
  $Prim \rightarrow \mathbf{BALL}$
  $Prim \rightarrow \mathbf{BOX}$
  $Prim \rightarrow \mathbf{NADA}$
  $Trans \rightarrow \mathbf{RX}\ Numero$
  $Trans \rightarrow \mathbf{RY}\ Numero$
  $Trans \rightarrow \mathbf{RZ}\ Numero$
  $Trans \rightarrow \mathbf{S}\ Numero$
  $Trans \rightarrow \mathbf{SX}\ Numero$
  $Trans \rightarrow \mathbf{SY}\ Numero$
  $Trans \rightarrow \mathbf{SZ}\ Numero$
  $Trans \rightarrow \mathbf{TX}\ Numero$
  $Trans \rightarrow \mathbf{TY}\ Numero$

[1] Python Lex Yacc

$Trans \rightarrow \textbf{TZ}\ Numero$
$Trans \rightarrow \textbf{CR}\ Numero$
$Trans \rightarrow \textbf{CG}\ Numero$
$Trans \rightarrow \textbf{CB}\ Numero$
$Trans \rightarrow \textbf{D}\ Numero$
$Numero \rightarrow Numero\ +\ Factor$
$Numero \rightarrow Numero\ -\ Factor$
$Numero \rightarrow Factor$
$Factor \rightarrow Factor\ *\ Termino$
$Factor \rightarrow Factor\ /\ Termino$
$Factor \rightarrow Termino$
$Termino \rightarrow \textbf{NUM}$
$Termino \rightarrow\ +\ \textbf{NUM}$
$Termino \rightarrow\ -\ \textbf{NUM}$
$Termino \rightarrow (\ Numero\ )$
$Termino \rightarrow\ +\ (\ Numero\ )$
$Termino \rightarrow\ -\ (\ Numero\ )$

De V_t cabe destacar el token **NADA** que es el que representa al caractér "_", el token **REGLA** que se corresponde con la expresión regular '[a-zA-Z]+' y el token **NUM** con la expresión regular '\d+(\.\d+)?'.
Los demás tokens se corresponden literalmente con su texto en minúscula, por ejemplo $BALL$ con la palabra "ball".
Para que estos últimos no sean tokenizados como **REGLA** se usó el ejemplo de palabras reservadas que aparece en la sección **4.3** de la documentación de ply (`http://www.dabeaz.com/ply/ply.html#ply_nn6`).
Existe también una forma de reconocer comentarios encerrados entre comillas dobles, los cuales serán descartados sin más.

3. Implementación de la solución

La solución consiste de dos grandes pasos.
El primero ocurre mientras se ejecuta el analizador sintáctico, en este paso se "sintetiza" en el **NT** que generó la producción el objeto que va a ser mostrado así también las transformaciones que sufre el mismo. Para las producciones donde se define un nombre de regla[2], por ejemplo "*bola = ball|box*", utilizamos dos diccionarios globales "reglas" y "finales". Si la regla no es final se define solo en el diccionario "reglas", en cambio si es final (tiene '.') se define en ambos diccionarios.
El segundo paso es al momento de mostrar, en este paso se toma del diccionario "reglas" la definición de **$** y se le ejecuta el método **mostrar** que se encargará de mostrar todos los elementos que se generaron durante el parseo.

Una estado se define con tres atributos:

- **space**: Consiste de una matriz de 4*4 de la cual se obtendrán los valores para ubicar el elemento en el espacio
- **color**: Un array de 3 posiciones que representan cada componente de color
- **depth**: Un entero que representa la máxima cantidad de reemplazos de reglas de la composición visual

[2]Estamos utilizando que $ es un nombre de regla

Transformar consiste en mezclar dos de estos estados de manera que el estado resultante sera un producto de matrices para **space**, un producto elemento a elemento para **color** y el menor de los **depth**.
De este modo podemos definir estado identidad como:
$$\textbf{space} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
$\textbf{color} = \begin{bmatrix} 1, 1, 1 \end{bmatrix}$
$\textbf{depth} = 100$

Para la solución se crearon dos jerarquías de clases que interactúan entre sí. Una es la que se refiere a las transformaciones:
De Transformacion heredan TransRX, TransRY, TransRZ, TransT, TransS, TransC, TransD, cada una de ellas representa un tipo de transformación según su nombre lo indica.
Cada transformación es un estado que al aplicarse a otro lo afecta de la manera deseada.
A su vez las instancias directas de Transformacion se corresponde con la transformación identidad, la cual al ser aplicada no efectúa ningún cambio. Ésta es el estado inicial de todo elemento.
La otra jerarquía es la de Elementos: De Elemento heredan Primitiva, Compuesta, ElementoPOT, ElementoCorchete y ElementoREGLA. De Primitiva heredan Ball, Box y Nada, y de Compuesta heredan ElementoOR, ElementoAND y ElementoANDTransformaDirecto.
Las clases del subárbol de Primitiva representan cada una un objeto final que se mostrará en la composición visual final.
Las clases del subárbol de Compuesta implementan un patrón "composite" los cuales reenviaran a sus contenidos los mensajes que reciban.
Las clases ElementoCorchete y ElementoPOT también siguen este patrón pero conteniendo un solo elemento.
La clase ElementoCorchete tiene mensajes especiales para interactuar con ElementoPOT, en caso de que ésta última lo contenga, en otro caso simplemente reenvía los mensajes a su elemento contenido.
Todos los Elementos tienen implementados los métodos para interactuar con ElementoPOT de forma tal que éste los repita sin el comportamiento especial que tiene ElementoCorchete.
Gracias a la forma en que está hecha la gramática, las precedencias están resueltas al momento del parseo, con lo cual, podemos ir creando los objetos a medida que el parser va interpretando el archivo.
ElementoOR representa una cadena de |, por ejemplo $A|B|C|D$ donde A, B, C, D puede ser cualquier elemento. Las producciones que parten del NT Elemento crean éste tipo de elemento de la siguiente manera:
$Elemento \rightarrow Elementoand$
$\{Elemento.valor \Leftarrow ElementoOR().append(Elementoand.valor\}$
$Elemento \rightarrow Elemento_1 \mid Elementoand$
$\{Elemento.valor \Leftarrow Elemento_1.valor.append(Elementoand.valor)\}$
Analogamente **ElementoAND** representa una cadena $A\&B\&C\&D$ y se creasiguiendo las mismas reglas.
ElementoRegla se crea instanciando con referencias a los diccionarios "reglas" y "finales' para las producciones:
$Elementobase \rightarrow$ **REGLA**
$Elementobase \rightarrow \$$
ElementoPOT y **ElementoCorchete** se crean respectivamente con las producciones:
$Elementobase \rightarrow Elementobase \; \hat{} \; Numero$
$Elementobase \rightarrow [\; Elemento \;]$
Con la produccion "$Elementobase \rightarrow \; < \; Elemento \; >$" se crea una instancia de **ElementoOR** donde se agrega el elemento junto con un elemento primitivo **Nada** para que tenga 50 % de

probabilidades de aparecer

Teniendo en cuenta que puede haber más de una definición del mismo nombre de regla y que esto es equivalente a tener un '|' entre las dos definiciones, todas las definiciones del diccionario reglas son **ElementoOR** al cual se van agregando las nuevas definiciones, teniendo así un único valor para la misma clave el cual se encargara de decidir cual de las definiciones aplicar[3].

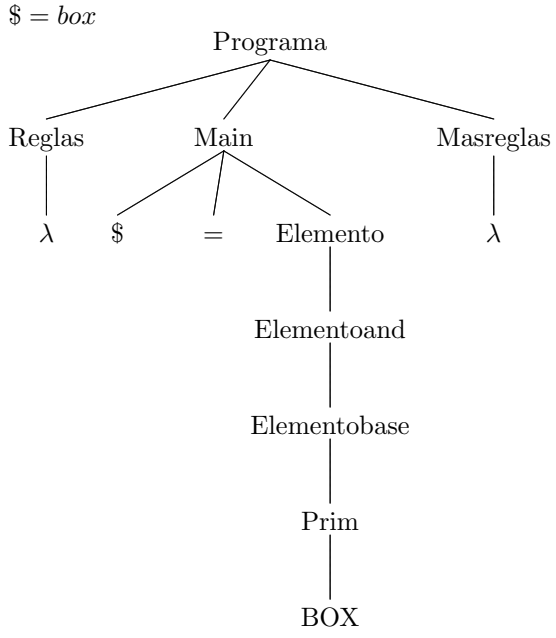
$$\left.\begin{array}{l} Reg = A \\ Reg = B \end{array}\right\} \equiv Reg = A|B \tag{1}$$

Los elementos tienen estados inicializados con la transformación identidad. Cada vez que reciben una transformación se modifica dicho estado.
Los estados tienen 2 mensajes principales: *transformar* que dada una Trasnformacion transforma su estado interno y *mostrar* que define las reglas de como se muestra el Elemento en la composición visual. Por ejemplo al mandarle el mensaje *mostrar* a un **ElementoOR**, éste tomará pseudo-aleatoriamente uno de sus elementos contenidos y le enviará el mensaje *mostrar*

4. Arboles de derivación

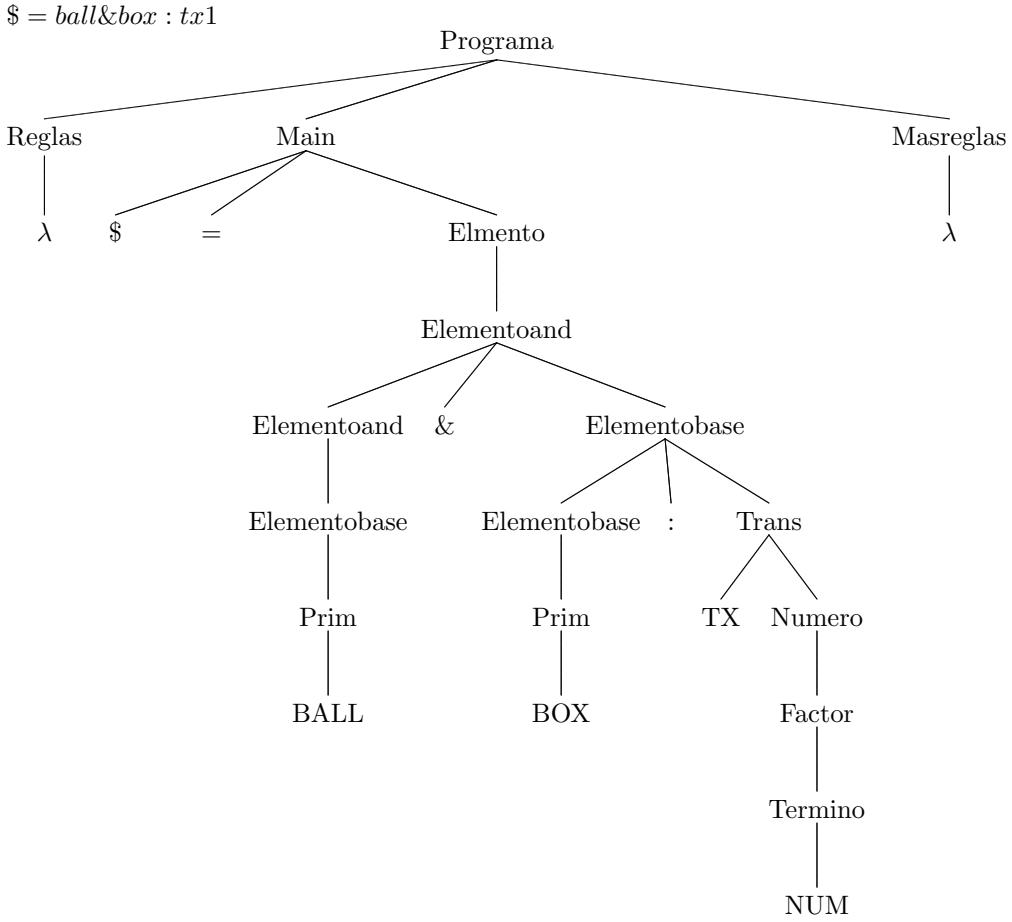
En esta sección se van a presentar algunos ejemplos parseados junto con los arboles de derivación que genera cada entrada.

- $\$ = box$

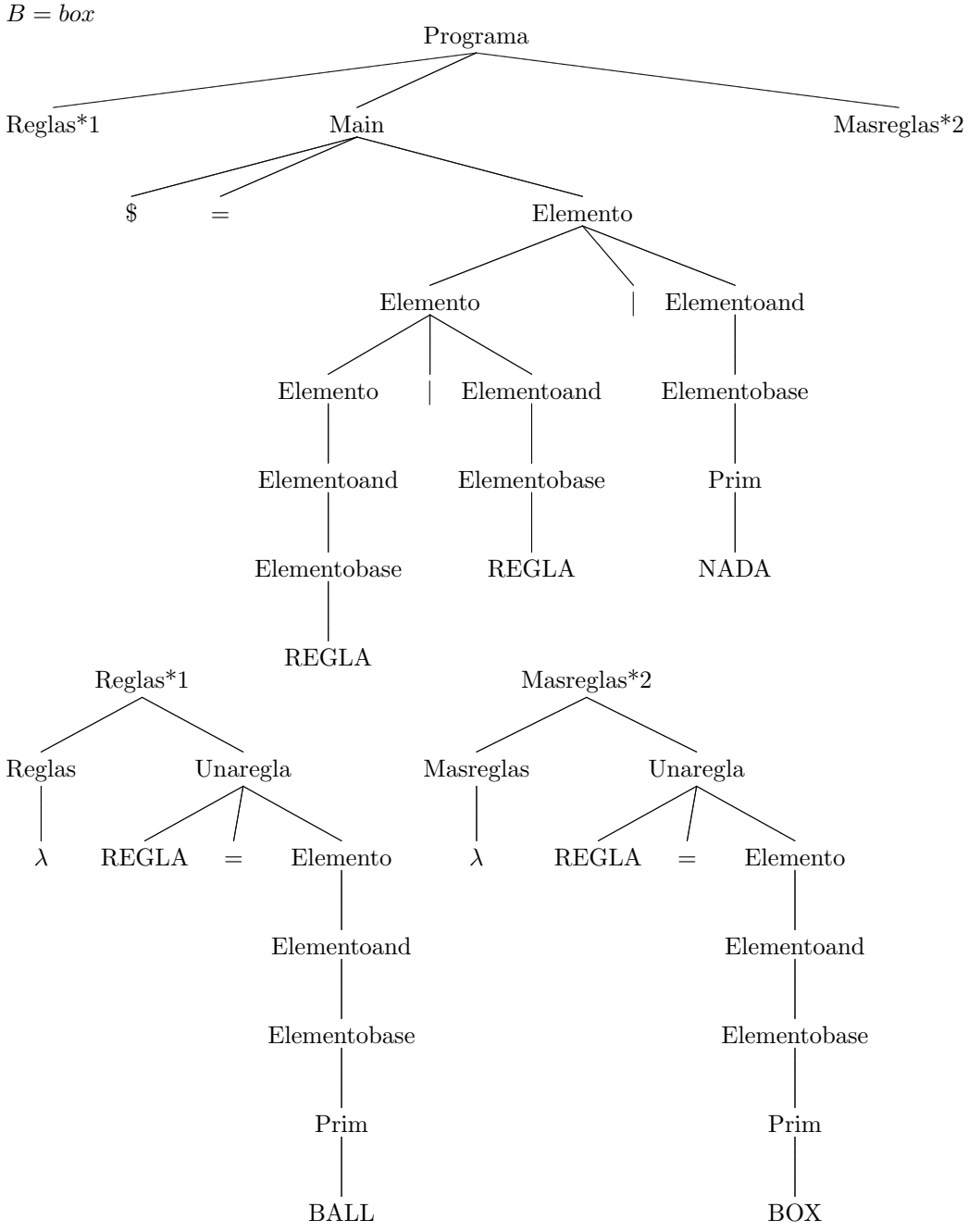


[3]Al ser tratada cada definición como Elementos diferentes, ésto no afecta a los '|' internos de cada definición

- $\$ = ball\&box : tx1$


Programa
Reglas
Main
Masreglas
λ
$
=
Elmento
λ
Elementoand
Elementoand
&
Elementobase
Elementobase
Elementobase
:
Trans
Prim
Prim
TX
Numero
BALL
BOX
Factor
Termino
NUM

- $A = ball$
  $\$ = A|B|\_$
  $B = box$



En el directorio Derivaciones se puede encontrar las listas de producciones utilizadas para generar cada uno de los ejemplos del directorio TL2014C2-TP-Ejemplos. Estas listas conforman las derivaciones mas a la derecha invertidas generadas por el parser LALR

5. Algunos resultados

5.1. Resultados válidos

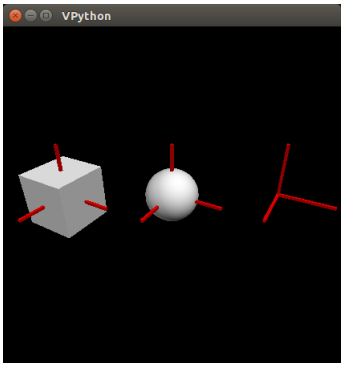


Figura 1: Primitivas eg01.peg

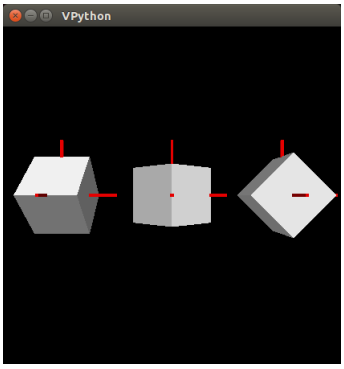


Figura 2: Transf. Rotar eg02.peg

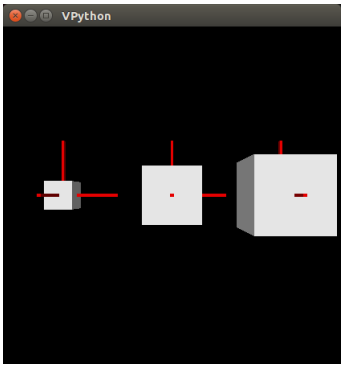


Figura 3: Transf. Escala simétrica eg03.peg

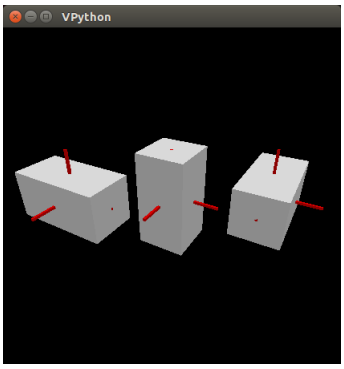


Figura 4: Transf. Escala en ejes eg04.peg

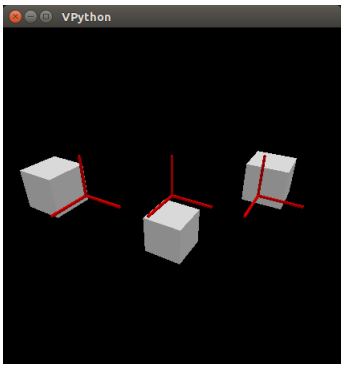


Figura 5: Transf. Traslación eg05.peg

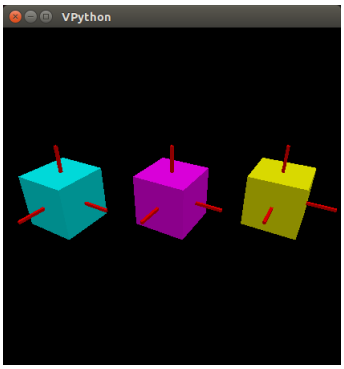


Figura 6: Transf. Color eg06.peg

5.2. Resultados inválidos

Hay tres tipos de resultados inválidos, de los cuales los dos primeros lanzan una excepción SyntaxError y el último una Exception :

- Caracteres inválidos en el archivo.
  $\$ = A|B$
  $Regla = ball\,\%$
  $A = B|Regla$
  El símbolo "%" no es parte del lenguaje
  SyntaxError: Caracter ilegal en la linea 2 cerca de '%
  A=B—Regla
  '

- Palabra mal formada.
  $\$ = A|B$
  $Regla55 = ball$
  $A = B|Regla55$
  Si bien el analizador léxico reconoce los números, los nombres de reglas sólo admiten letras mayúsculas y minúsculas.
  SyntaxError: Error de sintaxis en la linea 2 cerca de '55' reconocido como NUM '55.0'

- Otra forma de que el parser falle es si se utiliza alguna regla que no fue definida.
  $\$ = A|B$
  $Regla = ball$
  $A = B|Regla$
  $ usa las reglas A y B pero B nunca fue definida.
  Exception: La regla B no está definida

6. Ejecución

El programa debe ser ejecutado de la siguiente manera:
python parser.py ruta/al/archivo.peg

En el archivo parser.py hay tres variables boolenas globales útiles para mostrar más información:
mostrarTokens Imprime la lista de Tokens que generó el analizador léxico
mostrarEjes Muestra en la composición visual los ejes X(Rojo) Y(Verde) Z(Azul) centrado en el origen desde -5 a +5 en cada eje
mostrarProducciones Muestra la lista de producciones generada por el parser LALR

7. Detalle de requerimientos

Para la implementación del trabajo práctico se utilizaron las bibliotecas de Python:

- sys
- numpy
- math
- copy
- visual (VPython)

- random
- ply
- re

8. Desiciones

Decidimos pensar a $ como un nombre de regla cualquiera el cual puede estar definido varias veces e incluso puede ser llamado por otra regla; asimismo se le puede definir alguna final con "$.".

En las producciones
$Elementobase \rightarrow Elementobase\ :\ Trans$
$Trans \rightarrow \mathbf{RX}\ Numero$
$Trans \rightarrow \mathbf{RY}\ Numero$ (Por dar un ejemplo)
podríamos haber puesto
$Elementobase \rightarrow Elementobase\ :\ Trans\ Numero$
$Trans \rightarrow \mathbf{RX}$
$Trans \rightarrow \mathbf{RY}$
Decidimos por la primera porque nos permite crear el objeto de transformación instanciado en el mismo paso.

Optamos por esa estructura de clases para Elemento porque resultaba más natural delegarle a cada objeto que supiera sus reglas de mostrado y transformación, y la gramática nos dejaba armarlos transformarlos y componerlos fácilmente.
Probamos con otras producciones
$Programa \rightarrow Reglas\ Main\ Reglas$
$Reglas \rightarrow \lambda$
$Reglas \rightarrow Reglas\ Main$
$Reglas \rightarrow Reglas\ Unaregla$
Pero generaba un conflicto Reduce/Reduce yendo por Reglas → Main

$$\left.\begin{array}{c} Programa \rightarrow \bullet\ Reglas\ Main\ Reglas \\ Reglas \rightarrow \bullet \\ Reglas \rightarrow \bullet\ Reglas\ Main \end{array}\right\} \xrightarrow{Reglas} \left\{\begin{array}{c} Programa \rightarrow Reglas\ \bullet\ Main\ Reglas \\ Reglas \rightarrow Reglas\ \bullet\ Main \end{array}\right\} \xrightarrow{Main}$$

$$\left.\begin{array}{c} Programa \rightarrow Reglas\ Main\ \bullet\ Reglas \\ Reglas \rightarrow \bullet \\ Reglas \rightarrow Reglas\ Main\ \bullet \end{array}\right\}$$

9. Conclusión