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

Trabajo Práctico

Teoría de Lenguajes

Segundo Cuatrimestre de 2014

Apellido y Nombre	LU	E-mail
Delgado, Alejandro N.	601/11	nahueldelgado@gmail.com
Lovisolo, Leandro	645/11	leandro@leandro.me
Petaccio, Lautaro José	443/11	lausuper@gmail.com

1. Introducción

En este trabajo desarrollamos un visualizador para el formato de síntesis procedural PEGS, creado por el cuerpo docente de la materia Teoría de Lenguajes de nuestra facultad.

2. Especificación de la gramática

A continuación detallamos primero los tokens reconocidos por el lexer y luego las producciones reconocidas por el parser.

El formato PEGS fue extendido con tres nuevas primitivas: `cylinder`, `cone` y `torus`, que producen un cilindro de base y diámetro 1, un cono de base y altura 1 y un toro de diametro exterior 1, respectivamente.

2.1. Tokens

Las cadenas representadas por cada token se corresponden con su nombre en minúscula salvo especificado lo contrario.

Token	Descripción
`BOX`, `BALL`	Primitivas
`UNDERSCORE`	Primitiva vacía (carácter "_")
`CYLINDER`, `CONE`, `TORUS`	Primitivas adicionales
`RX`, `RY`, `RZ`	Transformaciones de rotación
`SX`, `SY`, `SZ`, `S`	Transformaciones de escala
`TX`, `TY`, `TZ`	Transformaciones de traslación
`CR`, `CG`, `CB`	Transformaciones de coloreo
`D`	Transformación de límite de profundidad de recursión
`COLON`	Inicio de una transformaciín (carácter ":")
`AND`, `OR`	Operaciones de conjunción y disyunción (caracteres "&" y "|")
`POWER`	Operación de potenciación (carácter "^")
`LGROUP`, `RGROUP`	Agrupación de operaciones (caracteres "[" y "]")
`LOPT`, `ROPT`	Operación opcional (caracteres "<" y ">")
`NUMBER`	Constante numérica (expresión regular `[0-9]+(\.[0-9]+)?`)
`PLUS`, `MINUS`, `TIMES`, `DIVIDE`	Operadores aritméticos binarios (caracteres "+", "-", "*" y "/")
`LPAREN`, `RPAREN`	Agrupación de operaciones aritméticas (caracteres "(" y ")")
`START_RULE`	Regla inicial (carácter "$")
`RULE`	Regla (expresión regular `[a-zA-Z]+`)
`DOT`	Regla final (carácter ".")
`EQUALS`	Declaración de regla (carácter "=")
`COMMENT`	Comentario (expresión regular `"([^\\\n]|(\\.))*"`)

2.2. Producciones

Las producciones a continuación están escritas en el formato consumido por la librería PLY.

La producción inicial de esta gramática es `rules`.

Notar que la producción `empty` no tiene cuerpo. Esto es porque, como su nombre señala, se corresponde con la producción vacía, comúnmente escrita $A \to \epsilon$ o $A \to \lambda$ en los libros de texto, y la librería PLY no provee ninguna sintaxis especial para tales producciones.

Notar además que algunas producciones utilizan una sintaxis especial `%proc`. Esta sintaxis se explica en la próxima sección.

```
rules : rule_definition rules
      | empty
```

```
rule_definition : RULE       EQUALS element
                | RULE DOT   EQUALS element
                | START_RULE EQUALS element

element : primitive
        | rule
        | transform
        | element_and
        | element_or
        | element_power
        | element_group
        | element_optional

primitive : BOX
          | BALL
          | UNDERSCORE
          | CYLINDER
          | CONE
          | TORUS

rule : RULE

transform : element COLON transform_name arith_expr

transform_name : RX
               | RY
               | RZ
               | SX
               | SY
               | SZ
               | S
               | TX
               | TY
               | TZ
               | CR
               | CG
               | CB
               | D

element_and : element AND element

element_or : element OR element

element_power : element POWER arith_expr

element_group : LGROUP element RGROUP

element_optional : LOPT element ROPT

arith_expr : arith_expr_number
           | arith_expr_uplus
           | arith_expr_uminus
           | arith_expr_parenthesis
           | arith_expr_plus
           | arith_expr_minus
           | arith_expr_times
           | arith_expr_divide
```

```
arith_expr_number : NUMBER

arith_expr_uplus : PLUS arith_expr %prec UPLUS

arith_expr_uminus : MINUS arith_expr %prec UMINUS

arith_expr_parenthesis : LPAREN arith_expr RPAREN

arith_expr_plus : arith_expr PLUS arith_expr

arith_expr_minus : arith_expr MINUS arith_expr

arith_expr_times : arith_expr TIMES arith_expr

arith_expr_divide : arith_expr DIVIDE arith_expr

empty :
```

2.3. Órdenes de precedencia y asociatividades

Las ambiguedades en las producciones anteriores se resuelven con los órdenes de precedencia y asociatividades en la tabla a continuación.

Notar que la tabla incluye dos tokens ficticios `UPLUS` y `UMINUS`. Éstos se corresponden con los operadores unarios "+" y "-", que definen el signo de una expresión aritmética. Notar que tienen un orden de precedencia superior al de sus contrapartes binarios.

Estos tokens ficticios se referencian desde las producciones `arith_expr_uplus` y `arith_expr_uminus`, que usan los tokens `PLUS` y `MINUS`, respectivamente, pero que en el contexto de esas producciones particulares requieren un orden de precedencia distinto. La librería PLY provee un mecanismo para cambiar el orden de precedencia de un token en una producción: escribiendo `%prec <TOKEN>` al final de una producción, el orden de precedencia del token usado en esa producción se reemplaza por el orden de precedencia del token `<TOKEN>` en ese contexto.

Se decidió darle un mayor orden de precedencia al token `POWER` por sobre los tokens `AND` y `OR`. Dicha precedencia no estaba clara en la especificación del lenguaje, y decidimos desambiguarlo guiados por nuestro gusto.

Orden	Token	Asociatividad	Comentarios
1	`PLUS`	Izquierda	Operadores aritméticos binarios
1	`MINUS`	Izquierda	
2	`TIMES`	Izquierda	
2	`DIVIDE`	Izquierda	
3	`UPLUS`	Derecha	Operadores aritméticos unarios
3	`UMINUS`	Derecha	
4	`AND`	Izquierda	Operadores binarios sobre elementos
5	`OR`	Izquierda	
6	`POWER`	Izquierda	Operadores unarios sobre elementos
7	`COLON`	Izquierda	

3. Implementación

La ejecución del programa se divide en dos fases: parsing de la entrada y rendering de la escena. Explicamos ambas a continuación.

3.1. Parsing

Durante esta fase se ejecuta el parser generado con la librería PLY y se arma una instancia de la clase `Scene`, que captura la semántica de la entrada del programa.

`Scene` tiene una variable de instancia `Scene::rules` que mantiene un arreglo de instancias de la clase `Rule` o alguna de sus subclases. Este arreglo tiene un item por cada regla definida en la entrada. Por ejemplo, la entrada a continuación genera un arreglo `Scene::rules` con 4 items:

```
a  = box  & a : tx 1
a  = ball & a : tx 1
a. = box : cg 0
$  = a : d 10
```

Las reglas iniciales y finales corresponden a las subclases `StartRule` y `FinalRule`. El resto a la clase `Rule`.

Cada regla se modela mediante un árbol de parsing con exactamente la misma estructura reconocida por el parser. Cada nodo de este árbol se representa con una instancia de alguna subclase de `Node`, definida de la siguiente manera:

```
class Node:
  def __init__(self, scene):
    self.scene = scene
    self.children = []
  ...
```

Notar que la clase `Rule` es subclase de `Node`. En consecuencia, cada instancia de `Rule` en el arreglo `Scene::rules` es el nodo raíz del árbol de parsing de alguna regla definida en la entrada.

Existe aproximadamente una subclase de `Node` por cada producción en la gramática. Por ejemplo, las producciones `element_and` y `element_or` se corresponden con las subclases `And` y `Or`, respectivamente.

La librería PLY permite asociar acciones semánticas a cada producción. Por ejemplo, la producción `arith_expr_number` está asociada a la siguiente función:

```
def p_arith_expr_number(self, t):
  'arith_expr_number : NUMBER'
  t[0] = scene.Number(self.scene, t[1])
```

Esta función asocia al resultado de reconocer esta producción (`t[0]`) una instancia de la clase `Number` (subclase de `Node`) que recibe el valor numérico del token `NUMBER` (`t[1]`).

De forma similar, la producción `arith_expr_plus` está asociada a la siguiente función:

```
def p_arith_expr_plus(self, t):
  'arith_expr_plus : arith_expr PLUS arith_expr'
  t[0] = scene.Plus(self.scene, t[1], t[3])
```

Notar que el lado derecho de esta producción incluye nodos no terminales. En este caso, los contenidos de `t[1]` y `t[3]` serán instancias de subclases de `Node` creadas por el código asociado a producciones `arith_expr`. La función termina asignándole a `t[0]` una instancia de `Plus`, subclase de `Node`, que tiene como hijos a las dos instancias de `Node` resultantes de los nodos no terminales.

El resto de los nodos de un árbol de parsing de una regla cuaqluiera se instancian de forma similar.

A continuación se muestra una entrada de ejemplo `eg09.peg`:

```
$ = box:tx -1 & ball:tx 1
```

Y su correspondiente árbol de parsing, obtenido pasándole el parámetro `-p` al comando `pegv`:

```
Rule "$"
  And
    TX
      Box
      UMinus
        Number "1.0"
    TX
      Ball
      Number "1.0"
```

3.2. Rendering

El rendering de la escena se hace por medio de la librería Panda3D. Ésta mantiene su propio árbol de objetos visibles en una escena.

Cada objeto tiene atributos posición, rotación y escala, todos sobre los ejes X, Y y Z. A su vez, un objeto puede tener subobjetos. Todos los atributos de un objeto hijo son relativos a los del objeto padre. De esta forma, basta con modificar los atributos de un objeto padre para afectar a los objetos hijos de la forma esperada intuitivamente.

Todas las subclases de `Node` (instanciadas en la fase de parsing) implementan un método `render()` que devuelve un objeto de Panda3D resultante de realizar la acción correspondiente a ése nodo. Por ejemplo, el método `Box::render()` devuelve un objeto de Panda3D correspondiente a un cubo de lado 1 centrado en el origen.

Un objeto de Panda3D devuelto por el método `render()` de una subclase de `Node` siempre está centrado en el origen. Esto facilita la aplicación de transformaciones sucesivas sin tener que calcular manualmente los atributos de los objetos hijos afectados.

En el caso que una instancia de `Node` tenga nodos hijos, al invocarse su método `render()`, ésta a su vez puede invocar al método `render()` de sus nodos hijos y/o crea un nuevo objeto de Panda3D. Este nuevo objeto de Panda3D puede o no tener como hijos a los objetos de Panda3D obtenidos de invocar al método `render()` de los nodos hijos. Algunos ejemplos:

- Una instancia de `RZ`, correspondiente a rotar su nodo hijo sobre el eje Z, al invocarse su método `render()`, primero obtiene un objeto de Panda3D resultante de invocar al método `render()` de su nodo hijo, luego modifica su atributo rotación sobre el eje Z y devuelve dicho objeto.
- Una instancia de `TX`, correspondiente a modificar la posición sobre el eje X de su nodo hijo, al invocarse su método `render()`, primero crea un nuevo objeto de Panda3D inicialmente vacío, luego le agrega como hijo al objeto de Panda3D obtenido de invocar al método `render()` de su nodo hijo, modifica la posición del objeto hijo sobre el eje X y devuelve el objeto padre.
- Una instancia de `And`, al invocarse su método `render()`, crea un nuevo objeto de Panda3D inicialmente vacío, al que le agrega como hijos los objetos de Panda3D resultantes de invocar al método `render()` de sus dos nodos hijos.

El resto de las subclases de `Node` actúan de forma similar.

Dos subclases de `Node` particulares que valen la pena aclarar son `RuleElement` y `Power`.

La subclase `RuleElement` modifica la instancia de `Scene` agregándole estado sobre la regla que se está renderizando actualmente y el nivel de recursión actual (ver el diccionario `Scene::perRuleDepths`.) Además, en el caso de una llamada recursiva, decide si llamar a una regla general o a una regla final de acuerdo a si se alcanzó el límite de recursión particular para esa regla o global.

La subclase `Power` logra el efecto deseado creando nuevas instancias fantasma de `Power` con menor potencia, aplicándoles a dichas instancias fantasma las transformaciones requeridas y renderizando el nuevo nodo fantasma.

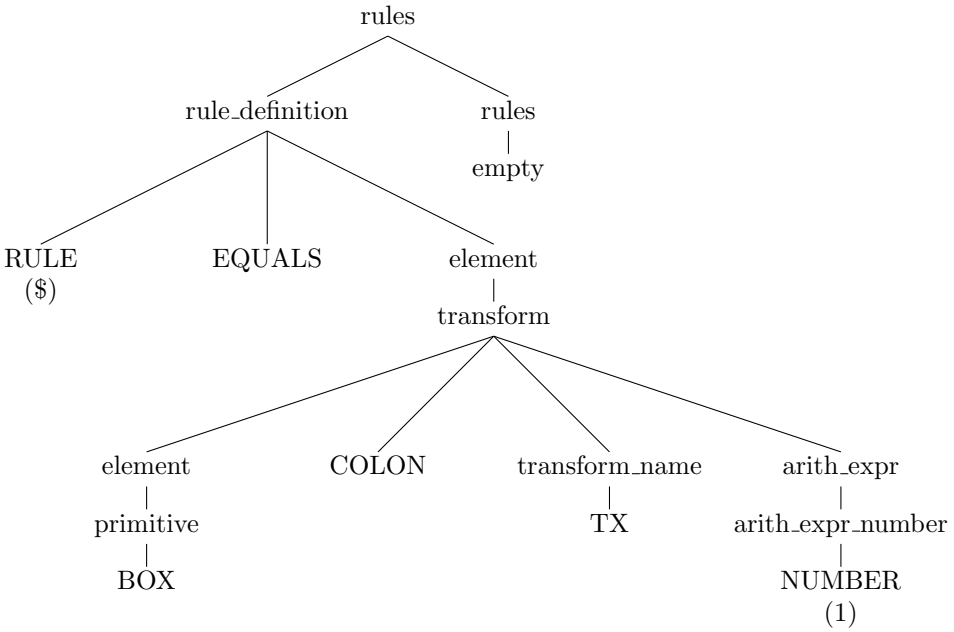
Finalmente, la escena entera se renderiza invocando el método `Scene::do_render()`, que simplemente obtiene la regla inicial "$" e invoca su método `render()`.

4. Ejemplos de árboles de derivación

Entrada:

```
$ = box : tx 1
```

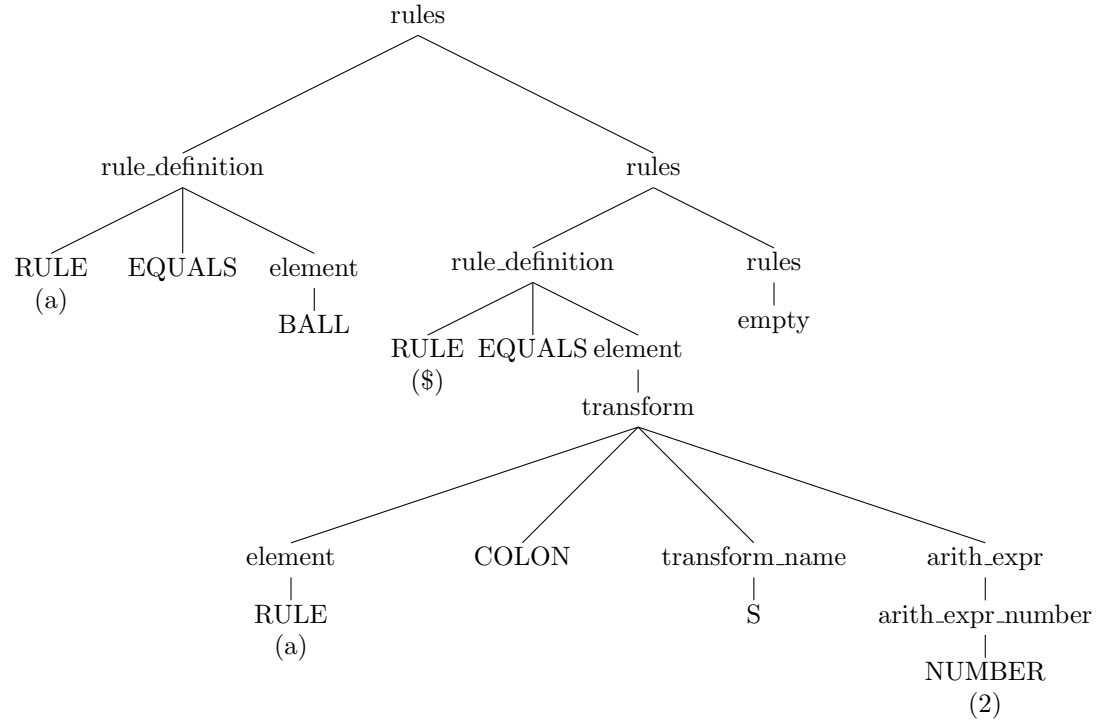
Árbol de derivación:



Entrada:

```
a = BALL
$ = a : s 2
```

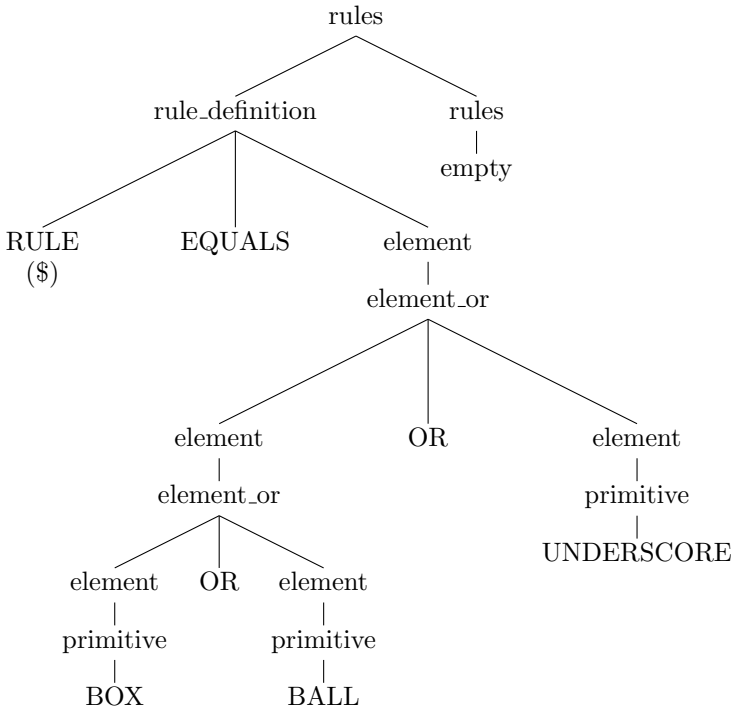
Árbol de derivación:



Entrada:

```
$ = box | ball | _
```

Árbol de derivación:



5. Ejemplos

Presentamos a continuación algunos ejemplos que ilustran la salida del visualizador.

5.1. Dick Butt

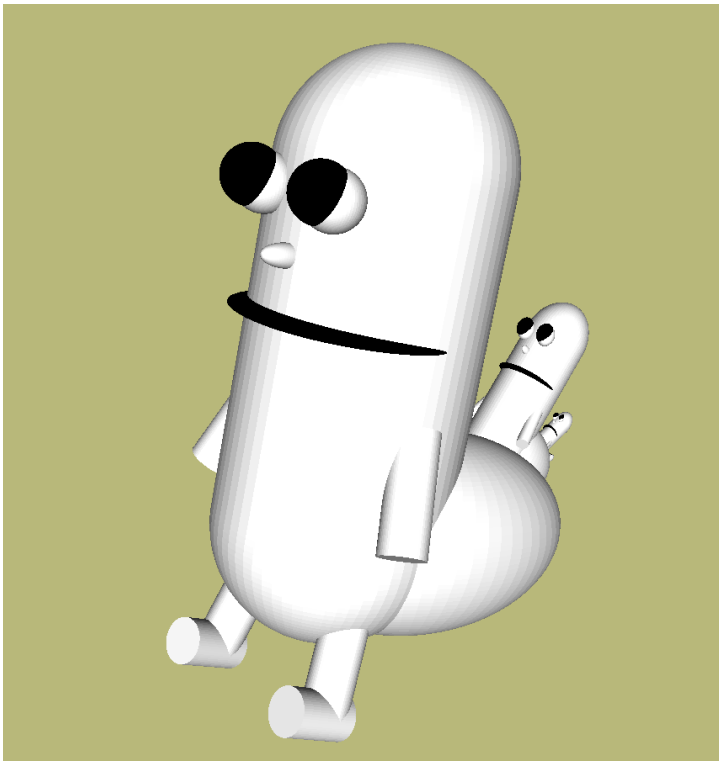
(Ver `http://knowyourmeme.com/memes/dick-butt`.)

Entrada:

```
$ = background
  & dickbutt : s 2 : ry 45 : rx 15

background = wall : tz -100
           & wall : tz  100
           & wall : ry 90 : tx  100
           & wall : ry 90 : tx -100
           & wall : rx 90 : ty  100
           & wall : rx 90 : ty -100


wall = box : s 1000 : sz 0.01 : cr 0.6 : cg 0.6 : cb 0.4

dickbutt = shaft
         & face : ty 0.125
         & butt
         & dickbutt : d 5 : s 0.3 : sy 1.3 : rz -30 : tx 1.15 : ty -0.2
         & legs
```

```
        & arms

face =
   "Mouth"
   torus : cr 0 : cg 0 : cb 0 : sy 0.2 : tx -0.1

   "Nose"
 & ball : s 0.1 : sx 2.5 : tx -0.5 : ty 0.25

   "Eyes"
 & ball : s 0.25 : tx -0.5  : ty 0.5  : tz -0.2
 & ball : s 0.25 : tx -0.55 : ty 0.52 : tz -0.22 : cr 0 : cg 0 : cb 0
 & ball : s 0.25 : tx -0.5  : ty 0.5  : tz  0.2
 & ball : s 0.25 : tx -0.55 : ty 0.52 : tz  0.18 : cr 0 : cg 0 : cb 0

shaft =
   cylinder : sy 1.5
 & ball : ty  0.75
 & ball : ty -0.75

butt =
   ball : s 1.5 : sy 0.6 : sz 0.5 : ry  10 : ty -0.75 : tx 0.35 : tz -0.2
 & ball : s 1.5 : sy 0.6 : sz 0.5 : ry -10 : ty -0.75 : tx 0.35 : tz  0.2

legs =
   leg : rz -15 : rx  10 : tx 0.2 : tz -0.1
 & leg : rz -15 : rx -10 : tx 0.2 : tz  0.1

leg =
   cylinder : s 0.2 : sy 2.5 : ty -1.25
 & cylinder : s 0.2 : sy 1.5 : rz 90 : ty -1.5 : tx -0.06125

arms = arm : rz -15 : rx  20 : tx 0.2 : ty 0.7 : tz -0.1
     & arm : rz -15 : rx -20 : tx 0.2 : ty 0.7 : tz  0.1

arm =
   cylinder : s 0.2 : sy 2.5 : ty -1.25
```

Resultado:

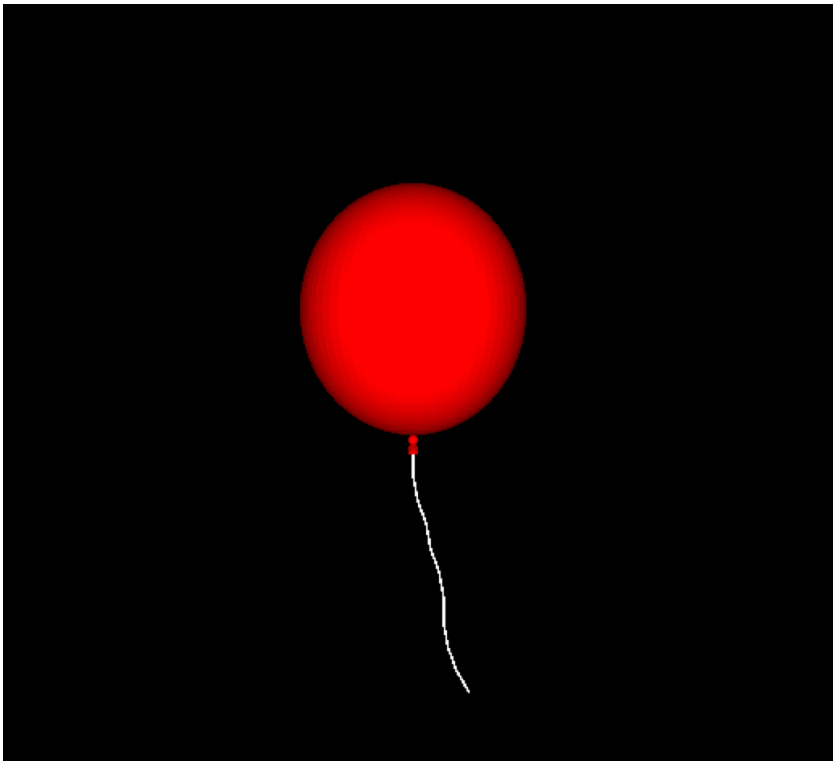


5.2. Globo

Entrada:

```
balloon = ball:cg0:cb0:sx0.9:sz0.9                  "el globo"
        & ball:cg0:cb0:s0.04:ty-0.52                "el nudo"
        & box:cg0:cb0:s0.03:rz45:rx45:ty-0.56       "el pico"
        & line:ty-0.56:d10                          "el hilo"

line = box:sx0.01:sy0.1:sz0.01:ty-0.05 & [ line:rz10:ty-0.1 | line:rz-10:ty-0.1 ]

$ = balloon
```

Resultado:



5.3. Programa inválido 1

Entrada:

```
$ = ball : ts 1
```

Salida:

```
Syntax error at line 1, column 12:
$ = ball : ts 1
           ^
```

El token `ts` es inválido. El parser detecta esto y reporta correctamente el error.

5.4. Programa inválido 2

Entrada:

```
ejes = box:sy0.05:sz0.05:cg0:cb0:tx0.5
$ = ojos : tx 2
```

Salida:

```
LookupError: Rule ojos not found.
```

La entrada es correcta desde el punto de vista gramático, sin embargo se hace referencia a una regla que no existe. El visualizador reporta dicho error y termina la ejecución.

6. Requerimientos

- Python 2.7
- Librería PLY (`http://www.dabeaz.com/ply/`)
- Librería Panda3D (`https://www.panda3d.org/`)

7. Modo de uso

Ejecutar `./pegv <archivo>` en el directorio raíz, reemplazando `<archivo>` por la ruta a algún archivo en formato PEGS. Ejemplo: `./pegv ejemplos/eg01.peg`.

Alternativamente proveer el código a interpretar por la entrada estándar. Por ejemplo: `cat ejemplos/eg01.peg | ./pegv`

Opcionalmente puede imprimirse cada regla reconocida y su respectivo árbol de parsing con la opción `-p`, por ejemplo: `./pegv -p ejemplos/eg01.peg`.

8. Código fuente

8.1. main.py

```
#!/usr/bin/env python2
# coding: utf-8

import argparse
import sys
from ply.lex import LexError
from lexer import Lexer
from parser import Parser

def parse(path, print_parse_tree):
  f = open(path)
  input = ''.join(f.readlines())
  f.close()

  try:
    parser = Parser(input)
  except LexError:
    sys.exit(-1)

  if not parser.has_errors:
    if print_parse_tree:
      parser.print_nodes()
    parser.scene.do_render()

if __name__ == '__main__':
  parser = argparse.ArgumentParser()
  parser.add_argument('path', metavar='FILE', default='/dev/stdin', nargs='?',
                      help='input file (default: read from standard input)')
  parser.add_argument('-p', action='store_true',
                      help='print parse tree to standard output')
  args = parser.parse_args()

  parse(args.path, args.p)
```

8.2. lexer.py

```
# coding: utf-8

import ply.lex as lex

class Lexer:

  keywords = (
    'BOX', 'BALL',                      # Primitivas
    'CYLINDER', 'CONE', 'TORUS',        # Primitivas de yapa
    'RX', 'RY', 'RZ',                   # Rotación
    'SX', 'SY', 'SZ', 'S',              # Escala
    'TX', 'TY', 'TZ',                   # Traslación
    'CR', 'CG', 'CB',                   # Color
    'D'                                 # Máxima profundidad
    )

  tokens = (
    'UNDERSCORE',                       # Primitiva nula
    'COLON',                            # Inicio de una transformación
    'AND', 'OR', 'POWER',               # Operaciones
    'LGROUP', 'RGROUP',                 # Agrupación de operaciones
    'LOPT', 'ROPT',                     # Operación opcional
    'NUMBER',                           # Constantes numéricas
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', # Operaciones aritméticas
    'LPAREN', 'RPAREN',                 # Agrupación de operaciones aritméticas
    'START_RULE', 'RULE', 'EQUALS',     # Reglas
    'DOT',                              # Regla final
    'COMMENT',                          # Comentarios
    'ID'                                # Keywords y nombres de regla
    ) + keywords

  # Keywords y nombres de regla
  def t_ID(self, t):
    r'[a-zA-Z]+'
    if t.value.upper() in Lexer.keywords:
      t.type = t.value.upper()
    else:
      t.type = 'RULE'
    return t

  # Primitivas
  t_UNDERSCORE    = r'_'

  # Inicio de una transformación
  t_COLON         = r':'

  # Operaciones
  t_AND           = r'&'
  t_OR            = r'\|'
  t_POWER         = r'\^'
  t_LGROUP        = r'\['
  t_RGROUP        = r'\]'
  t_LOPT          = r'<'
  t_ROPT          = r'>'

  # Números y aritmética
  t_NUMBER         = r'[0-9]+(\.[0-9]+)?'
  t_PLUS           = r'\+'
  t_MINUS          = r'-'
  t_TIMES          = r'\*'
  t_DIVIDE         = r'/'
```

```
  t_LPAREN         = r'\('
  t_RPAREN         = r'\)'

  # Reglas
  t_START_RULE     = r'\$'
  t_DOT            = r'\.'
  t_EQUALS         = r'='

  # Comentarios
  t_ignore_COMMENT = r'\"([^\\\n]|(\\.))*\"'

  # Caracteres ignorados
  t_ignore     = ' \t\n'

  def t_error(self, t):
    line = self.find_line(t)
    column = self.find_column(t)
    print 'Illegal character at line %d, column %d:' % (line, column)
    print self.input.split('\n')[line - 1]
    print ' ' * (column - 1) + '^'

  # Devuelve el número de línea del token dado
  def find_line(self, t):
    line = 0
    pos = t.lexpos
    while pos != -1:
      line += 1
      pos = self.input.rfind('\n', 0, pos)
    return line

  # Devuelve el número de columna del token dado
  def find_column(self, t):
    last_cr = self.input.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 lex(self):
    self.lexer.input(self.input)

  def __init__(self, input):
    self.input = input
    self.lexer = lex.lex(module=self)
```

8.3. parser.py

```
# coding: utf-8

from ply import yacc
from lexer import Lexer
import scene

class Parser:

  tokens = Lexer.tokens

  precedence = (
    ('left',  'PLUS',  'MINUS'),
    ('left',  'TIMES', 'DIVIDE'),
    ('right', 'UPLUS', 'UMINUS'),
    ('left',  'AND'),
    ('left',  'OR'),
```

```
  ('left',  'POWER'),
  ('left',  'COLON'),
  )

def p_rules(self, t):
  '''rules : rule_definition rules
           | empty'''
  if t[1] is not None:
    self.scene.rules.append(t[1])

def p_rule_definition(self, t):
  '''rule_definition : RULE       EQUALS element
                     | RULE DOT   EQUALS element
                     | START_RULE EQUALS element'''
  if t[1] == '$':
    t[0] = scene.StartRule(self.scene, t[3])
  elif t[2] == '.':
    t[0] = scene.FinalRule(self.scene, t[1], t[4])
  else:
    t[0] = scene.Rule(self.scene, t[1], t[3])

def p_element(self, t):
  '''element : primitive
             | rule
             | transform
             | element_and
             | element_or
             | element_power
             | element_group
             | element_optional'''
  t[0] = t[1]

def p_primitive(self, t):
  '''primitive : BOX
               | BALL
               | UNDERSCORE
               | CYLINDER
               | CONE
               | TORUS'''
  primitives = {'box'      : scene.Box,
                'ball'     : scene.Ball,
                '_'        : scene.Underscore,
                'cylinder' : scene.Cylinder,
                'cone'     : scene.Cone,
                'torus'    : scene.Torus}
  t[0] = primitives[t[1]](self.scene)

def p_rule(self, t):
  'rule : RULE'
  t[0] = scene.RuleElement(self.scene, t[1])

def p_transform(self, t):
  'transform : element COLON transform_name arith_expr'
  transforms = {'rx' : scene.RX,
                'ry' : scene.RY,
                'rz' : scene.RZ,
                'sx' : scene.SX,
                'sy' : scene.SY,
                'sz' : scene.SZ,
                's'  : scene.S,
                'tx' : scene.TX,
                'ty' : scene.TY,
                'tz' : scene.TZ,
```

```
                'cr' : scene.CR,
                'cg' : scene.CG,
                'cb' : scene.CB,
                'd'  : scene.D}
  element = t[1]
  transform_name = t[3]
  param = t[4]
  t[0] = transforms[transform_name](self.scene, element, param)

def p_transform_name(self, t):
  '''transform_name : RX
                    | RY
                    | RZ
                    | SX
                    | SY
                    | SZ
                    | S
                    | TX
                    | TY
                    | TZ
                    | CR
                    | CG
                    | CB
                    | D'''
  t[0] = t[1]

def p_element_and(self, t):
  'element_and : element AND element'
  t[0] = scene.And(self.scene, t[1], t[3])

def p_element_or(self, t):
  'element_or : element OR element'
  t[0] = scene.Or(self.scene, t[1], t[3])

def p_element_power(self, t):
  'element_power : element POWER arith_expr'
  t[0] = scene.Power(self.scene, t[1], t[3])

def p_element_group(self, t):
  'element_group : LGROUP element RGROUP'
  t[0] = scene.Group(self.scene, t[2])

def p_element_optional(self, t):
  'element_optional : LOPT element ROPT'
  t[0] = scene.Optional(self.scene, t[2])

def p_arith_expr(self, t):
  '''arith_expr : arith_expr_number
                | arith_expr_uplus
                | arith_expr_uminus
                | arith_expr_parenthesis
                | arith_expr_plus
                | arith_expr_minus
                | arith_expr_times
                | arith_expr_divide'''
  t[0] = t[1]

def p_arith_expr_number(self, t):
  'arith_expr_number : NUMBER'
  t[0] = scene.Number(self.scene, t[1])

def p_arith_expr_uplus(self, t):
  'arith_expr_uplus : PLUS arith_expr %prec UPLUS'
```

```
    t[0] = scene.UPlus(self.scene, t[2])

  def p_arith_expr_uminus(self, t):
    'arith_expr_uminus : MINUS arith_expr %prec UMINUS'
    t[0] = scene.UMinus(self.scene, t[2])

  def p_arith_expr_parenthesis(self, t):
    'arith_expr_parenthesis : LPAREN arith_expr RPAREN'
    t[0] = scene.Parenthesis(self.scene, t[2])

  def p_arith_expr_plus(self, t):
    'arith_expr_plus : arith_expr PLUS arith_expr'
    t[0] = scene.Plus(self.scene, t[1], t[3])

  def p_arith_expr_minus(self, t):
    'arith_expr_minus : arith_expr MINUS arith_expr'
    t[0] = scene.Minus(self.scene, t[1], t[3])

  def p_arith_expr_times(self, t):
    'arith_expr_times : arith_expr TIMES arith_expr'
    t[0] = scene.Times(self.scene, t[1], t[3])

  def p_arith_expr_divide(self, t):
    'arith_expr_divide : arith_expr DIVIDE arith_expr'
    t[0] = scene.Divide(self.scene, t[1], t[3])

  ##########################################################################

  def p_empty(self, t):
    'empty :'

  def p_error(self, t):
    line = self.lexer.find_line(t)
    column = self.lexer.find_column(t)
    print 'Syntax error at line %d, column %d:' % (line, column)
    print self.input.split('\n')[line - 1]
    print ' ' * (column - 1) + '^'
    self.has_errors = True

  def print_node(self, node, l=0):
    import sys
    sys.stdout.write('  ' * l)

    print node
    for i in range(0, len(node)):
      self.print_node(node[i], l+1)
    if isinstance(node, scene.Transform):
      self.print_node(node.param, l+1)

  def print_nodes(self):
    for r in self.scene.rules:
      self.print_node(r)

  def __init__(self, input):
    self.has_errors = False
    self.scene = scene.Scene()
    self.input = input
    self.lexer = Lexer(input)
    self.parser = yacc.yacc(module=self)
    self.parser.parse(input, lexer=self.lexer.lexer)
```

8.4. scene.py

```
# coding: utf-8

import os
from random import randint
from direct.showbase.ShowBase import ShowBase
from panda3d.core import *

################################################################################
# World                                                                        #
################################################################################

class World(ShowBase):
  def __init__(self):
    ShowBase.__init__(self)

    # Background color
    self.setBackgroundColor(0, 0, 0)

    # Directional light
    dlight = DirectionalLight('dlight')
    dlightnp = self.render.attachNewNode(dlight)
    dlightnp.setY(-100)
    self.render.setLight(dlightnp)

    # Ambient light
    ambient = AmbientLight('ambient')
    ambient.setColor(Vec4(0.2, 0.2, 0.2, 1))
    ambientnp = self.render.attachNewNode(ambient)
    self.render.setLight(ambientnp)

    # Camera position
    self.disableMouse()
    self.camera.setY(-20)
    mat = Mat4(self.camera.getMat())
    mat.invertInPlace()
    self.mouseInterfaceNode.setMat(mat)
    self.enableMouse()

################################################################################
# Scene                                                                        #
################################################################################

class Scene:
  def __init__(self):
    # Will be populated by the parser
    self.rules = []

    # Scene-wide maximum recursion depth
    self.maxDepth = 100

    # Current recursion depth
    self.currentDepth = 0

    # Rule => (depth, maxDepth) dictionary for rules with custom depth limits
    self.perRuleDepths = {}

    # Will be initialized by self.do_render()
    self.world = None

  def find_rule(self, name):
    criteria = lambda x : x.name == name
```

```
    rule = self.__find_rule_with_criteria__(criteria)
    if rule is None: raise LookupError('Rule %s not found.' % name)
    return rule

  def find_final_rule(self, name):
    criteria = lambda x : x.name == name and isinstance(x, FinalRule)
    rule = self.__find_rule_with_criteria__(criteria)
    return rule

  def __find_rule_with_criteria__(self, criteria):
    matching = filter(criteria, self.rules)
    if len(matching) == 0:
      return None
    else:
      return matching[randint(0, len(matching) - 1)]

  def new_detached_node(self):
    node = self.world.render.attachNewNode('node')
    node.setColor(1, 1, 1, 1)
    node.detachNode()
    return node

  def do_render(self):
    self.world = World()
    parent = self.find_rule('$').render()
    parent.reparentTo(self.world.render)
    self.world.run()

################################################################################
# Node                                                                         #
################################################################################

class Node:
  def __init__(self, scene):
    self.scene = scene
    self.children = []

  def __len__(self):
    return len(self.children)

  def __getitem__(self, index):
    return self.children[index]

  def __str__(self):
    return self.__class__.__name__

  def render(self):
    raise NotImplementedError()

################################################################################
# Rules                                                                        #
################################################################################

class Rule(Node):
  def __init__(self, scene, name, element):
    Node.__init__(self, scene)
    self.name = name
    self.children.append(element)

  def __str__(self):
    return 'Rule "%s"' % self.name

  def render(self):
```

```
    return self.children[0].render()

class FinalRule(Rule): pass

class StartRule(Rule):
  def __init__(self, scene, element):
    Rule.__init__(self, scene, '$', element)

################################################################################
# Arithmetic expressions                                                       #
################################################################################

class ArithExpr(Node):
  def value(self):
    raise NotImplementedError()

class Number(ArithExpr):
  def __init__(self, scene, value):
    ArithExpr.__init__(self, scene)
    self._value = float(value)

  def value(self):
    return self._value

  def __str__(self):
    return 'Number "%s"' % str(self._value)

class UnaryOp(ArithExpr):
  def __init__(self, scene, child):
    ArithExpr.__init__(self, scene)
    self.children.append(child)

class UPlus(UnaryOp):
  def value(self):
    return self.children[0].value()

class UMinus(UnaryOp):
  def value(self):
    return -1 * self.children[0].value()

class BinaryOp(ArithExpr):
  def __init__(self, scene, left, right):
    ArithExpr.__init__(self, scene)
    self.children.append(left)
    self.children.append(right)

class Plus(BinaryOp):
  def value(self):
    return self.children[0].value() + self.children[1].value()

class Minus(BinaryOp):
  def value(self):
    return self.children[0].value() + self.children[1].value()

class Times(BinaryOp):
  def value(self):
    return self.children[0].value() * self.children[1].value()

class Divide(BinaryOp):
  def value(self):
    return self.children[0].value() / self.children[1].value()

class Parenthesis(ArithExpr):
```

```
  def __init__(self, scene, child):
    ArithExpr.__init__(self, scene)
    self.children.append(child)

  def value(self):
    return self.children[0].value()

################################################################################
# Elements                                                                     #
################################################################################

class Element(Node): pass

class Box(Element):
  def render(self):
    node = loader.loadModel(os.path.dirname(__file__) + os.path.sep + 'box')
    def setColor(node):
      node.setColor(1, 1, 1, 1)
    nmap(setColor, node)
    return node

class Ball(Element):
  def render(self):
    node = loader.loadModel(os.path.dirname(__file__) + os.path.sep + 'ball')
    def setColor(node):
      node.setColor(1, 1, 1, 1)
    nmap(setColor, node)
    return node

class Underscore(Element):
  def render(self):
    return self.scene.new_detached_node()

class Cylinder(Element):
  def render(self):
    node = loader.loadModel(os.path.dirname(__file__) + os.path.sep + 'cylinder')
    def setColor(node):
      node.setColor(1, 1, 1, 1)
    nmap(setColor, node)
    return node

class Cone(Element):
  def render(self):
    node = loader.loadModel(os.path.dirname(__file__) + os.path.sep + 'cone')
    def setColor(node):
      node.setColor(1, 1, 1, 1)
    nmap(setColor, node)
    return node

class Torus(Element):
  def render(self):
    node = loader.loadModel(os.path.dirname(__file__) + os.path.sep + 'torus')
    def setColor(node):
      node.setColor(1, 1, 1, 1)
    nmap(setColor, node)
    return node

class RuleElement(Element):
  def __init__(self, scene, name):
    Element.__init__(self, scene)
    self.name = name

  def __str__(self):
```

```
    return 'RuleElement "%s"' % self.name

  def render(self):
    # Get per-rule recursion limit and current depth
    if self.name in self.scene.perRuleDepths.keys():
      hasPerRuleDepth = True
      currentPerRuleDepth, maxPerRuleDepth = self.scene.perRuleDepths[self.name]
    else:
      hasPerRuleDepth = False

    # Check if recursion limit has been reached
    if self.scene.currentDepth < self.scene.maxDepth and \
       (not hasPerRuleDepth or currentPerRuleDepth < maxPerRuleDepth):

      # Increase recursion depth
      if hasPerRuleDepth:
        currentPerRuleDepth += 1
        self.scene.perRuleDepths[self.name] = (currentPerRuleDepth, maxPerRuleDepth)
      self.scene.currentDepth += 1

      # Debug information
      # if hasPerRuleDepth:
      #   perRuleStr = "\tRule depth: %d/%d." % (currentPerRuleDepth, maxPerRuleDepth)
      # else: perRuleStr = ""
      # print "Rendering rule %s.\tGlobal depth: %d/%d.%s" \
      #     % (self.name, self.scene.currentDepth, self.scene.maxDepth, perRuleStr)

      # Render final rule if we're at the last recursive call and there is one
      if (self.scene.currentDepth == self.scene.maxDepth or \
          (hasPerRuleDepth and currentPerRuleDepth == maxPerRuleDepth)) and \
         self.scene.find_final_rule(self.name) is not None:
        result = self.scene.find_final_rule(self.name).render()

      # Not the last recursive call or no final rule
      else:
        result = self.scene.find_rule(self.name).render()

      # Decrease recursion depth
      self.scene.currentDepth -= 1
      if hasPerRuleDepth:
        currentPerRuleDepth -= 1
        self.scene.perRuleDepths[self.name] = (currentPerRuleDepth,  maxPerRuleDepth)

    # Recursion limit reached
    else:
      result = self.scene.new_detached_node()

    return result

##############################################################################
# Transforms                                                                 #
##############################################################################

class Transform(Element):
  def __init__(self, scene, child, param):
    Element.__init__(self, scene)
    self.children.append(child)
    self.param = param

class RX(Transform):
  def render(self):
    node = self[0].render()
    node.setP(node.getP() + self.param.value())
```

```
    return node

class RY(Transform):
  def render(self):
    node = self[0].render()
    node.setH(node.getH() + self.param.value())
    return node

class RZ(Transform):
  def render(self):
    node = self[0].render()
    node.setR(node.getR() - self.param.value())
    return node

class SX(Transform):
  def render(self):
    node = self[0].render()
    node.setSx(node.getSx() * self.param.value())
    return node

class SY(Transform):
  def render(self):
    node = self[0].render()
    node.setSz(node.getSz() * self.param.value())
    return node

class SZ(Transform):
  def render(self):
    node = self[0].render()
    node.setSy(node.getSy() * self.param.value())
    return node

class S(Transform):
  def render(self):
    node = self[0].render()
    node.setSx(node.getSx() * self.param.value())
    node.setSy(node.getSy() * self.param.value())
    node.setSz(node.getSz() * self.param.value())
    return node

class TX(Transform):
  def render(self):
    parent = self.scene.new_detached_node()
    node = self[0].render()
    node.setX(node.getX() + self.param.value())
    node.reparentTo(parent)
    return parent

class TY(Transform):
  def render(self):
    parent = self.scene.new_detached_node()
    node = self[0].render()
    node.setZ(node.getZ() + self.param.value())
    node.reparentTo(parent)
    return parent

class TZ(Transform):
  def render(self):
    parent = self.scene.new_detached_node()
    node = self[0].render()
    parent.setY(node.getY() - self.param.value())
    node.reparentTo(parent)
    return parent
```

```
class ColorTransform(Transform):
  def render(self):
    def change_color(node):
      r, g, b, a = node.getColor()
      rt, gt, bt = self.get_color_transform()
      node.setColor(r * rt, g * gt, b * bt, a)

    node = self[0].render()
    nmap(change_color, node)
    return node

  def get_color_transform(self):
    raise NotImplementedError()

class CR(ColorTransform):
  def get_color_transform(self):
    return (self.param.value(), 1, 1)

class CG(ColorTransform):
  def get_color_transform(self):
    return (1, self.param.value(), 1)

class CB(ColorTransform):
  def get_color_transform(self):
    return (1, 1, self.param.value())

class D(Transform):
  def render(self):
    child = self[0]
    while isinstance(child, Transform):
      child = child[0]

    if not isinstance(child, RuleElement):
      print "Ignoring maximum depth transform applied to non-rule element."
      return self[0].render()

    rule = child
    if rule.name not in self.scene.perRuleDepths.keys():
      self.scene.perRuleDepths[rule.name] = (0, self.param.value())
      result = self[0].render()
      self.scene.perRuleDepths.pop(rule.name)
    else:
      result = self[0].render()

    return result

################################################################################
# Operations                                                                   #
################################################################################

class And(Element):
  def __init__(self, scene, left, right):
    Element.__init__(self, scene)
    self.children.append(left)
    self.children.append(right)

  def render(self):
    parent = self.scene.new_detached_node()
    left  = self.children[0].render()
    right = self.children[1].render()
    left.reparentTo(parent)
    right.reparentTo(parent)
```

```
    return parent

class Or(Element):
  def __init__(self, scene, left, right):
    Element.__init__(self, scene)
    self.children.append(left)
    self.children.append(right)

  def render(self):
    choices = self.flatten()
    chosen = choices[randint(0, len(choices) - 1)]
    return chosen.render()

  def flatten(self):
    result = []
    if isinstance(self[0], Or): result += self[0].flatten()
    else:                       result.append(self[0])
    if isinstance(self[1], Or): result += self[1].flatten()
    else:                       result.append(self[1])
    return result

class Power(Element):
  def __init__(self, scene, child, power):
    Element.__init__(self, scene)
    self.children.append(child)
    self.power = power

  def render(self):
    # Base cases
    if self.power.value() < 1:
      return self.scene.new_detached_node()
    elif self.power.value() == 1:
      return self[0].render()

    # Recursive case: recursively create and render a new ghost Power node with
    # a smaller power, then render the child node
    else:
      # New Power node with smaller power
      new_power = Power(self.scene, self[0],
                        Number(self.scene, self.power.value() - 1))

      # Accumulate a list of transforms until the first element node is reached
      transforms = []
      child = self[0]
      while isinstance(child, Transform):
        transforms.append((child.__class__, child.param))
        child = child[0]

      # Apply transform chain to each conjunction if element node is a group
      if isinstance(child, Group):
        and_node = And(self.scene, new_power, child[0])
        node = self.build_transform_chain(transforms, and_node)

      # Ignore transforms list otherwise
      else:
        node = And(self.scene, new_power, self[0])

      return node.render()

  def build_transform_chain(self, transforms, node):
    while len(transforms) > 0:
      transform_class, param = transforms.pop()
      node = transform_class(self.scene, node, param)
```

```
    return node

class Group(Element):
  def __init__(self, scene, child):
    Element.__init__(self, scene)
    self.children.append(child)

  def render(self):
    return self.children[0].render()

class Optional(Element):
  def __init__(self, scene, child):
    Element.__init__(self, scene)
    self.children.append(child)

  def render(self):
    if randint(0, 1) == 0:
      return self.scene.new_detached_node()
    else:
      return self[0].render()

################################################################################
# Helper code                                                                  #
################################################################################

# Map function for Panda3D nodes
def nmap(f, node):
  f(node)
  for child in node.getChildren():
    nmap(f, child)
```