Universidad de Castilla-La Mancha
Escuela Superior De Informática

Procesadores de Lenguajes

# Blendify: Lenguaje para la definición de simulaciones físicas

*Autores:*
Alberto Aranda García
Cristian Gómez Portes
Daniel Pozo Romero
Eduardo Sánchez López

*Fecha:*
27 de enero de 2018

# Índice

# Índice de tablas

# 1. Presentación del problema

Dentro de la enseñanza de Física, la visualización de los problemas es una herramienta muy importante para que los estudiantes comprendan mejor los conceptos teóricos existentes tras ellos. La teoría del aprendizaje de Ausubel[1] propone este enfoque, ya que permite al alumno esclarecer y dotar de utilidad aquello que está aprendiendo [1]. En este contexto, la simulación es una opción que lleva disponible durante años, pero que está siendo muy infrautilizada.

Esto tiene una fácil explicación: la mayoría de programas de generación de simulaciones físicas imponen interfaces gráficas que requieren una inversión de tiempo grande por parte del usuario: partiendo de un enunciado textual de un problema, el usuario, que puede ser tanto un profesor como un estudiante de física, se ve obligado a traducirlo manualmente a un formato gráfico, lidiando muchas veces con interfaces de usuario tediosas que hacen perder mucho tiempo, tanto para aprenderlas, como por las operaciones repetitivas que deben realizarse.

Lo ideal sería que, a partir de un enunciado de un problema, fuese posible la generación automática de su simulación. Dejando de lado las dificultades que el Procesado de Lenguaje Natural supone (aún dentro de un dominio tan acotado), una primera aproximación puede ser la creación de un lenguaje con un poder expresivo similar al de los enunciados de problemas, al que sea lo más sencillo posible traducir (para un humano) dichos enunciados escritos originalmente en lenguaje natural.

Como plataforma de simulaciones físicas, vamos a considerar Blender, ya que incorpora, entre muchas otras funcionalidades, un motor de físicas muy potente, y, tantoo más importante, una API para Python que expone prácticamente todas las funcionalidades del programa.

Para nuestra práctica, por tanto, consideramos la definición de un lenguaje que permita escribir simulaciones físicas, y la construcción de un procesador de lenguajes que lo traduzca a código Python que use la API de Blender, que posteriormente podrá ser interpretado por este programa para generar una simulación. Nuestro objetivo principal, pues, es simplificar al máximo posible la generación de simulaciones que de otra forma serían muy tediosas de realizar.

[1] Psicólogo y pedagogo estadounidense de gran importancia para el constructivismo

# 2. Descripción del lenguaje

Como se ha dicho anteriormente, nuestro lenguaje, llamado **Blendify**, deberá tener el mismo poder expresivo los enunciados de problemas, y posiblemente, permitir definir ciertos hechos sobre el contexto que se dan por supuesto (tipo de problema, dimensionalidad, etc.). Si analizamos la estructura de los enunciados, nos encontramos con que, primero, se nos da información sobre los elementos que intervienen en el problema (cubo, esfera, polea, rampa, electrón, planeta...) y sus parámetros (masa, velocidad, fuerzas que actúan sobre este...); posteriormente, se suele dar una condición que podemos considerar como de parada de la simulación, y una pregunta (que podemos felizmente obviar) sobre algún parámetro de un elemento en determinado momento. Nuestro lenguaje, pues, deberá permitir declarar diferentes objetos y sus correspondientes atributos, así como ciertos parámetros globales que afecten a toda la simulación, alguna condición de parada, y posiblemente, información sobre qué quiere hacer el usuario con la simulación (ejecutarla, generar una animación, etc.).

## 2.1. Tabla de Tokens y Palabras Reservadas

En esta sección se muestra la tabla de *tokens* y palabras reservadas que componen le vocabulario de nuestro lenguaje — ver Tabla 1 y 2. Esta tabla nos ayudará a formar las producciones que generarán nuestro lenguaje final. En el siguiente apartado se podrá ver como se ha hecho uso de estos *tokens* en las diferentes producciones.

| **Tokens** | **Lexeme** | **Pattern** |
|---|---|---|
| Comma | , | , |
| Open Bracket | { | { |
| Closed Bracket | } | } |
| Open parenthesis | ( | ( |
| Closed parenthesis | ) | ) |
| Coordinates | (1.2, 1.2, 1.2) | ‘(’ real ‘,’ real ‘,’ real ‘)’ |
| Semicolon | ; | ; |
| Real | 23.21 | -?(([0-9]+)—([0-9]*\.[0-9]+)) |
| ID | variableV1 | [a-z]+[a-zA-Z0-9_]* |

Tabla 1: Tabla de tokens.

| Key Words | Lexeme | Pattern |
|---|---|---|
| begin | begin | begin |
| end | end | end |
| declaration | declaration | declaration |
| action | action | action |
| start_simulation | start_simulation | start_simulation |
| static | static | static |
| dynamic | dynamic | dynamic |
| speed | speed | speed |
| weight | weight | weight |
| scale | scale | scale |
| rotation | rotation | rotation |
| location | location | location |
| form figure | Cube, Sphere, Cone Cylinder, Force_field, Ramp | Cube \| Sphere \| Cone Cylinder \| Force_field \| Ramp |

Tabla 2: Tabla de palabras reservadas.

## 2.2. EBNF

Una vez tenemos definidos nuestros *tokens* y palabras reservadas, podemos comenzar a especificar nuestro lenguaje. Como producción de partida se ha pensado en una que contenga terminales simples para que el usuario (normalmente destinado a aquellos que no tiene mucha experiencia en el campo de la programación) no tenga dificultades a la hora de crear diferentes escenarios, ya que la herramienta está pensada para ayudar en el campo de la física. La producción sería la siguiente:

*program* ::= **begin** id body_program **end**.

Se pretende que el usuario define el comienzo y el fin del programa para ayudarle a no cometer errores que puedan entorpecer el desarrollo del mismo.

*body_program* ::= **declaration** body_declaration **action** body_action.

Como se ha mencionado anteriormente, el objetivo de esta producción sería el mismo, ya que hay una parte de declaración de variables, otra de escenarios y acciones.

*body_declaration* ::= **'{'** static_declaration static_dynamic **'}'** | **'{'** dynamic_declaration static_dynamic **'}'**.

Esta producción definine los campos de declaraciones. Básicamente, lo que haremos

será definir un tipo, el cual irá asociado a un identificador. Por ejemplo, en el caso de *body_declration* definiremos si las variables son estáticas (objetos que tendrán posición, rotación y escala) o dinámicas (objetos que tendrán velocidad y peso, además de las mencionadas anteriormente), además de especificar su tipo e identificador.

*body_action* ::= **'{' start_simulation ';' '}'**.

En *body_action* podremos ejecutar una serie de acciones que modifiquen los objetos de nuestra escena. Para esta versión la única acción que tendrá será la de iniciar la simulación.

*static_declaration* ::= static_object figure_declaration **'{'** attribute_declaration **'}'**.

*dynamic_declaration* ::= dynamic_object figure_declaration { value_object }.

Estas dos ultimas producciones son aquellas que definen los tipos de figuras que podremos especificar: Cube, Sphere, Cone, Cylinder, Force_field, Ramp y Plane. Además, el identificador que será posible concretar serán aquellos que comiencen por letra, sin distinción entre minúsculas y mayúsculas, seguido de letras o números.

*type_figure* ::= **Cube** | **Sphere** | **Cone** | **Cylinder** | **Force_field** | **Ramp** | **Plane**.

Estas tres ultimas producciones son aquellas que definen los tipos de figuras que podremos especificar: **Cube**, **Sphere**, **Cone**, **Cylinder**, **Force_field**, **Ramp**, **Plane**. Además, el identificador que será posible concretar serán aquellos que comiencen por letra, sin distinción entre minúsculas y mayúsculas, seguido de letras o números.

*figure_declaration* ::= type_figure id.

*figure_declaration* especifica el tipo de la declaración junto con el *id* que determina el nombre de la variable.

*attribute_declaration* ::= { value_static }.

Establece los valores estáticos para un determinado objecto. Permite tanto no tener ningún valor hasta declarar los tres tipos definidos. En caso de que no se estableciera ninguno, se añadirían valores por defecto.

*value_object* ::= { value_static } | { value_dynamic }.

*value_static* ::= [position] [rotation] [scale].

*value_dynamic* ::= [weight] [speed].

Estas tres últimas producciones permiten escoger cualquier valor, tanto dinámico como estático. Esto se debe a que si la declaración es dinámica, el objeto que se cree

pueda tener cualquier valor.

*value_static* y *value_dynamic* serán aquellas producciones que permitan definir si un objeto será estático, es decir, no podrá moverse, o dinámico, tendrá una velocidad y peso asociados.

*location* ::= **location** coordinates.

*rotation* ::= **rotation** coordinates.

*scale* ::= **scale** coordinates.

*weight* ::= **weight** real **';'**.

*speed* ::= **speed** coordinates.

Con estas 5 producciones se intenta definir la posición, rotación, escala, peso y velocidad de un objeto en un escenario específico. Todas ellas se formarán mediante coordenadas (conjunto de 3 valores definidos por los ejes 'x', 'y', 'z' especificado en *coordinates*) menos el peso, que será un número real.

*coordinates* ::= **'('** real **','** real **','** real **')'**.

# 3. Jflex y Cup

## 3.1. Léxico

Para la parte léxica hemos usado *JFlex*, detectando expresiones regulares para definir los terminales del lenguaje que usaremos más adelante. El funcionamiento básico consiste en que cuando se detecta una de las palabras claves, Jflex retornará un objeto de la clase Symbol para que en el análisis sintáctico se trate de la manera correspondiente. Es en el análisis léxico donde tenemos la posibilidad de dar opción a que pueda haber comentarios en nuestra gramática además de informar de error si se ha escrito en el archivo fuente algún símbolo desconocido. El procedimiento para ignorar los comentarios en CUP es recogiendo los tokens que JFlex retorna y utilizarlos como terminales. Consiste en descartar todo lo que este entre medias de '/*' y '*/', incluyendo dichos símbolos. Para ello, una vez que encontramos '/*', pasamos a un estado "Comment", el cuál no retorna ningún símbolo ni hace ningún tipo de análisis léxico.

## 3.2. Sintáctico

Esta parte es en la cual se recogen los *tokens* que el analizador léxico ha reconozido. Cup es el encargado de analizar y verificar que los símbolos que recibe están estructurados de una manera correcta mediante un conjunto de producciones. Además, se encarga de detectar errores para informar de que alguna declaración está mal construida o que los valores no son los correctos. Tiene la posibilidad de recuperación, pero en caso de no ser posible, mandará un mensaje a la salida de error para informar al usuario en la línea y columna donde se produjo tal error.

## 3.3. Semántico

Dado que el objetivo de este proyecto es poder generar código *python* a través de nuestra gramática inventada, hemos diseñado un pequeño prototipo para *CUP* en el cual crearemos código para la creación de los objetos estáticos. Básicamente se trata de una prueba de concepto con la cuál demostramos nuestros conocimientos sobre dicha materia, la cuál también es escalable para un proyecto completo y finalizado ya que solo se trataría de añadir acciones a nuestras reglas semánticas ya escritas y diseñadas.

# Referencias

[1] David Ausubel et al. Teoría del aprendizaje significativo. *Fascículos de CEIF*, 1:1–10, 1983.