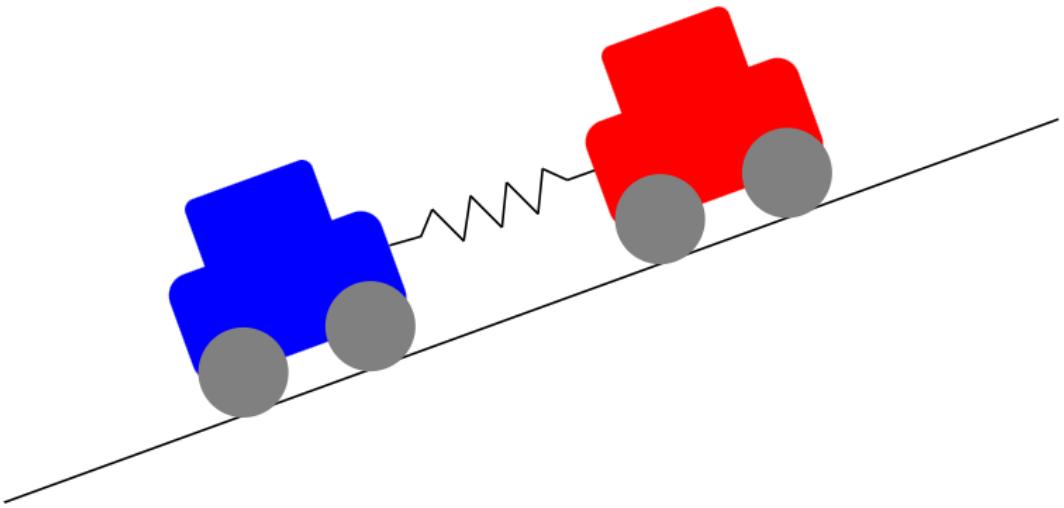
Autómatas, Teoría de Lenguajes y Compiladores

TPE: Newtonian - Grupo 5



Profesores: Mario Agustín Golmar y Ana María Arias Roig

Alumnos: Mauro Leandro Báez (61747), Franco David Rupnik (62032), Matías Manzur (62498) y Federico Shih (62293)

Fecha de Entrega: 22 de junio de 2023

Tabla de Contenidos

Introducción

El presente informe describe el desarrollo de nuestro proyecto Newtonian, un compilador el cual traduce el código siguiendo una sintaxis creada por nosotros en un archivo SVG. Permitiéndole a todo aquel que maneje nuestro lenguaje, Newtonian, generar distintos tipos de diagramas comunes en ejercicios de Física Newtoniana.

A grandes rasgos, este documento se concentrará en cómo fue pensada la sintaxis y cómo llevamos a cabo la implementación. Se puede dividir el desarrollo en tres partes:

1) La idea y creación de la Gramática la cual define la sintaxis a seguir en nuestro lenguaje.
2) El desarrollo del frontend, en el cual se describió cómo se leen las distintas partes del código, considerando la gramática y así asegurándose la correctitud de la sintaxis del código introducido. Distinguiendo entre lo que consideramos un archivo con código bien formado (el cual sigue la gramática) y un archivo con código que no sigue las reglas de sintaxis.
3) El desarrollo del backend, ya pudiendo discriminar entre código correctamente formado y código incorrecto, esta etapa se encarga de la generación del archivo SVG con la imagen correcta a partir del código introducido.

Consideraciones

Para una descripción más detallada del funcionamiento de los distintos componentes del lenguaje, leer la documentación de Newtonian en el archivo adjunto en el repositorio.

Se tiene que considerar que, los ejercicios que se pueden realizar con nuestro lenguaje son aquellos que incluyan planos inclinados, horizontales y verticales junto con bloques/autos y cilindros/bolas conectados por cuerdas, flechas y resortes. Decidimos limitarnos a estos objetos ya que son los más comunes a encontrarse cuando uno está creando ejercicios para un curso de Física Newtoniana básica.

Una consideración importante sobre los términos que se usarán en este informe es el uso de “padre” e “hijo”. A esto se refiere a que un “Objeto A” puede ser padre de otro cuando éste encierra con “{ }” a otro objeto. Si llamamos “Objeto B” a este, entonces se podría decir que “Objeto B” es hijo de “Objeto A” y “Objeto A” es padre de “Objeto B”. En cambio, si “Objeto B”,

a su vez, es padre de “Objeto C” entonces “Objeto C” no se considera hijo de “Objeto A”.

Otra consideración es el uso de columnas y filas para facilitar el alineamiento de objetos en un plano vertical u horizontal respectivamente. Las filas alinearán objetos “horizontalmente”, mientras que las columnas alinearán objetos de forma “vertical” (el uso de comillas viene porque dependiendo del contexto, no siempre el alineamiento será vertical u horizontal, leer la documentación para un mayor entendimiento). En ambos casos los objetos estarán seguidos uno del otro, es decir si se quiere dejar un espacio, se tendrá que usar un objeto especial llamado Spacer.

Cabe destacar que no se pueden anidar columnas o filas de forma directa, es decir una columna o fila no puede ser padre de otra columna o fila. En cambio si tenemos una fila que es padre de un bloque, que a la vez el bloque es padre de otra columna, este escenario si es permitido de recrear con nuestro lenguaje.

Descripción de la Gramática y Sintaxis

La idea principal fue poder manejar una sintaxis similar a la de Kotlin Compose (lenguaje utilizado para desarrollo de aplicaciones Android), para poder definir objetos de distinto tipo los cuales cuentan con propiedades y además pueden tener más objetos adentro.

Dado un objeto, lo que esté dentro de sus paréntesis, ‘(‘ y ‘)’, serán sus propiedades. La sintaxis de las mismas tiene la forma de: nombre de la propiedad, seguido por ‘:’ y el valor que uno pretende que esa propiedad tenga.

Dado un objeto, todo lo que esté dentro de sus llaves, ‘{‘ y ‘}’, sus objetos hijos, son aquellos que están relacionados directamente con el padre. Como por ejemplo una Arrow dentro de un Block, dicho Arrow aparecerá directamente ligada al Block.

Dado un objeto, si después de los paréntesis y antes de las llaves, existe un ‘:’ lo que le sigue es el valor de su posición. Esta no es simplemente una propiedad más por algo que describiremos más adelante en este informe.

Existen también los alignment (row y column), los cuales le ayudan al usuario a mejorar cómo están organizados los distintos objetos dentro de un esquema.

Decisiones de Diseño: Frontend

En un principio, “position” era una de las tantas propiedades que tenía un objeto, la cual define la posición donde se dibujará el objeto con respecto al padre. Esta propiedad no estaba presente si el padre era una columna/fila, ya que siempre se dibujan de la misma manera los objetos (de abajo hacia arriba e izquierda a derecha), o si era el primer objeto en el código, ya que no tiene padre para referenciar.

Si queríamos verificar en la gramática esta diferencia en la propiedad “position”, que podía llegar a tener un objeto dependiendo del padre, teníamos que dividir los objetos en dos versiones: una cuando el padre era una fila/columna o era el primero en el código y una cuando no lo era. Además de esto, se repetían las propiedades en común de ambos casos por lo que se volvió muy verborrágico.

Luego de que en la primera entrega se haya marcado este último hecho, se decidió cambiar la sintaxis al uso de un tag especial luego de la definición de las propiedades y el nombre del objeto, si es que el objeto era hijo de otro que no sea una fila/columna.

De esta forma seguimos teniendo que dividir en dos casos la gramática, pero esta división se puede hacer mucho antes de llegar a las producciones relacionadas con las propiedades de los objetos, reduciendo en gran medida la cantidad de producciones repetidas (o similares salvo por pequeños detalles) que tenía la gramática.

Otra medida que tomamos luego de la primera entrega es agrupar propiedades que se repetían en varios objetos a un general properties y luego las propiedades específicas para cada objeto. Esto mismo se hizo para reducir la cantidad de propiedades repetidas en las producciones de cada objeto aún más.

Cabe destacar que estas decisiones se hicieron con el fin de reducir la cantidad de cosas que debía revisar el backend, ya que al separar en dos casos los objetos con position y position-less, no fue necesario hacer uso de scopes para revisar el uso de position. También, al limitar las propiedades que tiene un objeto en la gramática, conseguimos que siempre las propiedades que lleguen al backend sean las apropiadas para cada objeto. Además, también revisamos los tipos de las propiedades, es decir que siempre será int, float o string en cada caso que corresponda. Por ejemplo, una label nunca llegará como un int al backend.

Finalmente, se decidió definir bison actions genéricas para construir el AST, de tal forma de seguir la naturaleza recursiva de la generación del árbol. Esto ayudó a que sólo se tuvieran que definir 5 funciones, una para el

programa, una para un objeto, para el conjunto de propiedades, para el cuerpo de un objeto (sus hijos) y finalmente para las propiedades solas.

Decisiones de Diseño: Backend

Mientras estábamos confeccionando la parte del front end, contábamos constantemente con una duda crucial, cómo íbamos a pasar de nuestro código a una imagen con las cualidades especificadas en el código.

Estuvimos barajando utilizar JavaFX para que el usuario pueda después compilar el código y obtener la imagen. Teníamos pensado hardcodear determinadas clases y después vincularlas, pero no habíamos llegado ni a revisar alguna librería de JavaFX cuando se nos ocurrió investigar acerca de SVG.

Generar nuestra propia librería de SVG terminó siendo mucho más cómodo que intentando usar Java, pues usamos como inspiración una librería que encontramos en la red que servía para realizar dibujos básicos en SVG (Ver referencias). Con esto pasamos a definir funciones que procesan los nodos del árbol de forma recursiva usando las funciones definidas en la librería ya mencionada.

Otra decisión importante fue la de no recorrer dos veces el árbol para buscar errores y warnings, sino que ya en la etapa de creación del AST, a medida que se van creando los nodos, vamos mirando si existe algo que si bien está habilitado por la gramática, conlleva a un error a la hora de la generación de código. Sin embargo, la mayoría de los errores los terminamos detectando con la gramática, por lo que no hace falta revisar tantas cosas durante la creación del AST. Esto presenta un trade off, ya que sin duda la gramática terminó siendo más compleja de lo que pudo haber sido si no hacíamos cosas como chequeos de tipos en la misma.

Problemas Encontrados

El principal problema fue el pensar la gramática para no tener que crear dos versiones de cada objeto con una de ellas que sí acepta una propiedad de posición y una que no, lo cual fue mencionado anteriormente en las decisiones de diseño de Frontend.

Analizar posibles errores antes de la generación de código presentaba una complicación, sin embargo lo pudimos resolver, durante la generación del AST, como fue descrito anteriormente.

El único problema vinculado con SVG (el cual seguramente también hubiese estado presente de haber usado JavaFX), fue el calcular a través de funciones trigonométricas, los ángulos necesarios para rampas y para rotaciones, ya que involucran a la propiedad en cuestión, su objeto y, muchas veces, a su padre también.

Limitaciones y Futuras Extensiones

La principal limitación es la imposibilidad de utilizar poleas las cuales son comunes en los tipos de ejercicios de Física que Newtonian viene a graficar. Creemos que nuestra gramática fue lo suficientemente compleja como para lo que apunta este trabajo práctico aún sin las poleas, esto mismo fue consultado con la cátedra la cual estuvo de acuerdo.

También pensamos en que se podrían agregar pequeñas funcionalidades como por ejemplo que las label permitan notación común en ejercicios de Física como una F con una pequeña flecha para explicitar la Fuerza. O funcionalidades como tener un offset absoluto para permitirle mayor personalización al usuario. Además de poderle cambiar el color a más cosas, personalizar tamaño de letras, etc.

Conclusión

En conclusión, con este proyecto hemos logrado entender la funcionalidad básica de un compilador usando Flex y Bison, junto con el diseño de un lenguaje completo con su sintaxis y gramática. Además, el uso de SVG como formato de salida ha demostrado ser una elección bastante buena, pues facilitó la generación de imágenes y dio la oportunidad de aprender una nueva herramienta para futuros proyectos.

Aunque existen limitaciones, el proyecto presenta un gran potencial de mejora y expansión en futuras versiones.

Bibliografía y Referencias

https://www.codedrome.com/svg-library-in-c/

Flex & Bison, O'Reilly https://web.iitd.ac.in/~sumeet/flex__bison.pdf

https://developer.mozilla.org/en-US/docs/Web/SVG