

# **72.39 Autómatas, Teoría de Lenguajes y Compiladores**

## **TPE**



(62028) Nicolás Matías Margenat - [nmargenat@itba.edu.ar](mailto:nmargenat@itba.edu.ar)

(62094) Juan Burda - [jburda@itba.edu.ar](mailto:jburda@itba.edu.ar)

(62493) Saul Ariel Castañeda - [scastaneda@itba.edu.ar](mailto:scastaneda@itba.edu.ar)

(62504) Elian Paredes - [eparedes@itba.edu.ar](mailto:eparedes@itba.edu.ar)

# Tabla de contenidos

---

<b>1. Introducción</b>	<b>2</b>
<b>2. Desarrollo del proyecto</b>	<b>2</b>
2.1. Frontend	2
2.2. Backend	2
<b>3. Dificultades encontradas</b>	<b>4</b>
<b>4. Futuras extensiones</b>	<b>4</b>
<b>5. Conclusión</b>	<b>5</b>
<b>6. Críticas constructivas para futuros TPs</b>	<b>5</b>
<b>7. Referencias</b>	<b>6</b>

# 1. Introducción

---

El objetivo del trabajo fue diseñar y desarrollar un lenguaje de programación junto con su compilador utilizando los conocimientos adquiridos en la materia. El lenguaje desarrollado buscó simplificar la manera de generar animaciones complejas mediante una sintaxis simple e intuitiva. Se utilizó Anime.js [1], una biblioteca de animaciones de Javascript, para simplificar la generación de dichas animaciones; y “uthash” [2] para el hash map que se utiliza tanto en la tabla de símbolos como en los parámetros que se guardan dentro de los nodos de los distintos objetos. En la etapa de generación de código se utilizó una biblioteca para concatenar strings llamada “C-StringBuilder” [3], la cual facilitó la generación de código CSS, HTML y JavaScript.

## 2. Desarrollo del proyecto

---

### 2.1. Frontend

El desarrollo del trabajo comenzó con el planteamiento y diseño del lenguaje. Para ello, el equipo decidió basarse en la sintaxis de Jetpack Compose, un kit de herramientas moderno de Android que utiliza una sintaxis declarativa para compilar IU nativas.

Dentro de las decisiones importantes que se tomaron en esta fase fue decidir qué parámetros, tanto de CSS como de Anime.js [1], admitirían los objetos del lenguaje. Se optó finalmente por mantener un set reducido pero robusto de ellos, permitiendo al usuario combinarlos para obtener efectos más complejos. Este aspecto se discutió en la primera entrega del trabajo y no sufrió cambios significativos por lo que no se hará ningún comentario adicional.

En lo que se refiere a los tipos de datos (es decir, las figuras, animaciones, vectores y layouts) es importante notar que, desde el primer informe hasta la implementación, se eliminó la opción de tener como layout una “grid”. La causa de ello fue la complejidad de implementarla, así como también la falta de tiempo por parte del equipo. De todos modos, se la deja como una de las futuras extensiones del lenguaje.

En cuanto a las animaciones, si se compara el primer informe con la implementación final, se puede notar la falta de aquellas que denominamos “presets”. Los presets no fueron implementados, de nuevo, por falta de tiempo por parte del equipo. Si bien su implementación no era difícil, se optó por utilizar ese tiempo para lograr una mayor calidad y claridad de código.

### 2.2. Backend

Lo primero que se realizó durante esta fase fue la construcción del árbol de sintaxis abstracta. En un principio se optó por seguir el *approach* presentado en clase, que consistía en generar un nodo por cada símbolo no terminal, y hacer que cada producción tenía su propia acción de Bison. El resultado fue un código inmenso e inmantenible, por lo que se decidió consultar con el profesor acerca de alternativas para el mismo. De las opciones discutidas,

aquella que más beneficios traería sería la que usaba *unions* y *enums* para distinguir entre las distintas producciones. Más adelante en el proyecto, el equipo se dio cuenta que podría simplificarse todavía más el código, dejando solo unos pocos structs que podrían ser distinguidos por el generador de código, y que permitirían la correcta generación del mismo. Esto, sin embargo, fue algo en lo que el equipo reparó en la siguiente etapa: la creación de la tabla de símbolos.

La tabla de símbolos fue esclarecedora en muchos aspectos, ya que fue aquí donde se evidenció qué cosas eran realmente necesarias y cuáles no. Para empezar, muchos de los nodos que se tenían eran innecesarios, y podían ser agrupados en sus tipos más elementales. Por ejemplo, no era necesario tener un “OpacityAnimationNode” pues podría distinguirse dentro de una “Animation” por su tipo dentro de “AnimationType”. Asimismo, los parámetros podían simplificarse enormemente metiéndolos a todos en una misma *union*, y juntando todos sus tipos en un *enum*. De esta manera, serían distinguibles entre sí, y sabríamos cómo acceder a ellos por su tipo de dato real (es decir, si bien podríamos decir que un “text-decoration” puede tomar los valores “italic”, “bold”, etc. en realidad estos son strings, por lo que podrían ser accedidos de esa manera). Consecuentemente, la cantidad de código se redujo drásticamente, y permitió que seguirlo no fuese una tarea extremadamente tediosa, y programarlo una tarea monótona y repetitiva.

En cuanto a los scopes de las variables y la coerción de tipos, estas no tienen sentido en un lenguaje como Bubblegum. Los scopes tendrían sentido si hubiese variables que se pueden crear anidadas dentro de las estructuras, cosa que no es posible dentro del lenguaje. Asimismo, la coerción de datos no tiene sentido pues las variables están para ser asignadas una vez y no ser reasignadas; esto es así porque el código se ejecuta una única vez (al final), y no hay operaciones que necesiten de variables auxiliares. La principal razón por la que se incorporaron las variables al lenguaje es para hacer el código más legible y reducir su repetición.

Para la generación de código se utilizó la biblioteca “C-StringBuilder” [2] y se crearon 3 *string builders*, uno para cada archivo generado (es decir, uno para el código HTML, uno para CSS y uno para JS). La complejidad de esta parte del código vino dada por relacionar las figuras, animaciones y layouts en el CSS y JS. Esto se terminó haciendo de una manera no del todo elegante, dado que siempre que hay una animación se genera un bloque de Anime.js, en lugar de chequear si la animación es hija de otra, y consecuentemente agregarle las propiedades de la nueva a la animación padre. De todos modos, se cree que esto no dificulta el entendimiento del código generado. Finalmente, es importante notar que el programa, cuando se lo corre desde la terminal, genera los 3 archivos que si se ponen un server corren y muestran la animación correctamente.

Con el objetivo de concretar la construcción de un editor web que permita programar y ver las animaciones en tiempo real, se agregó la capacidad de generar una versión del compilador en Web Assembly y Javascript. Para lograr esto, se utilizó la librería Emscripten [4], que proporciona las herramientas necesarias para completar dicho proceso a partir del proyecto en C utilizando LLVM. Se aprovechó la configuración del proyecto con CMake y se parametrizó adecuadamente, para mantener simplificado el proceso de compilación para ambos entornos de ejecución. Actualmente, dependiendo de los parámetros utilizados, se

generará un módulo Javascript del compilador que expone las funciones necesarias para transformar el código de nuestro lenguaje a una animación dentro de un entorno web.

Finalmente, un aspecto importante que se debe mencionar es que se decidió dejar de dar soporte al tipo de dato Text. Esta decisión se tomó luego de ver las animaciones en acción y notar que no aportaban ningún aspecto novedoso. De todos modos, durante la discusión que se tuvo sobre si sacarla o no, surgieron ideas que se podrían implementar a futuro, como por ejemplo hacer que cada letra de una palabra cambie de color, o hacer que se muevan en forma de onda. Es por esta razón que se dejó el código para aquella futura implementación

### 3. Dificultades encontradas

---

La dificultad principal que tuvo el equipo a lo largo del proyecto fue mantener la cantidad de código repetido a un mínimo. Esto fue un gran desafío dada la naturaleza del lenguaje, que recibe listas de parámetros extensas y que admiten un orden arbitrario. Este problema se resolvió extrayendo toda la lógica común a estructuras lo más arriba en el árbol de sintaxis posible, y hacer uso de los tipos de dato *union* y *enum* que brinda C. Este es un punto que se viene repitiendo a lo largo del informe, pero la realidad que supuso un antes y un después el comprender cómo funcionaban y podían ser usados estos tipos de datos. En materias anteriores rara vez son mencionados, pero aquí se terminó de comprender el valor que representan.

Una dificultad menor fue, en un principio, comprender el funcionamiento de Flex y Bison, y la manera en la que ambos se comunicaban. Esto se resolvió leyendo la bibliografía recomendada por la cátedra (el libro de Flex y Bison de Levine [5]).

### 4. Futuras extensiones

---

Las extensiones que se le pueden realizar al proyecto son diversas:

- La extensión más obvia que se puede hacer es aceptar una mayor cantidad de parámetros para los objetos. Esto brindaría mayor libertad creativa al usuario y enriquecería muchísimo lo que se puede lograr con él.
- Agregar más tipos de imágenes, como por ejemplo *svg*, podría realizarse con facilidad dada la manera en la que se realizaron las producciones dentro de Bison para los datos de tipo Media.
- El tipo de layout “grid” no se implementó por falta de tiempo. Este tipo de dato permitiría definir celdas en las que se podrían colocar distintas animaciones, dándole al usuario una manera extremadamente simple de distribuir objetos en la pantalla. De todos modos, es posible lograr un comportamiento idéntico al de un grid, anidando layouts de “center”, “vertical” y “horizontal”, pero el tipo “grid” simplificaría este trabajo.
- Otra extensión que se podría realizar sobre el proyecto sería implementar una biblioteca de animaciones prehechas (los presets que se mencionaron en el primer

informe). De esta manera, el usuario tendría un catálogo más amplio de opciones, así como también una idea de lo que se puede lograr utilizando Bubblegum.

- Permitir exportar las imágenes como *gif* o videos, así como también exportar la animación como un componente que pueda ser insertado directamente en una página Web.
- Implementar la línea de tiempo y el *staggering*, dos funcionalidades que provee Anime.js para poder sincronizar el movimiento de varios objetos con gran facilidad. Asimismo, se podrían incorporar los *keyframes* para simplificar la traslación de las imágenes.
- Implementar autocompletado y formateo de código para mejorar la experiencia del desarrollador.
- Implementar animaciones de texto complejas, como por ejemplo colorear letras de una palabra dinámicamente, hacer que la palabra “ondee”, etc.

## 5. Conclusión

---

Este trabajo le permitió al equipo aprender muchísimo, no solo de Flex y Bison, sino también de características de C que no se ven muy seguido. Similarmente, permitió aplicar los conocimientos teóricos vistos en clase de una manera integradora.

En retrospectiva, el equipo está muy sorprendido de lo que logró hacer: no solo el lenguaje, sino también hacer la página que permite editar el código y ver el resultado al instante. Esta fue una aspiración que se tuvo desde que se concibió la idea, pero había grandes dudas respecto a la posibilidad de lograr hacerla.

## 6. Críticas constructivas para futuros TPs

---

En esta sección se dejará de hablar de una manera tan impersonal, y vamos a pasar a hablar más humanamente :). Esperamos que no nos baje la nota.

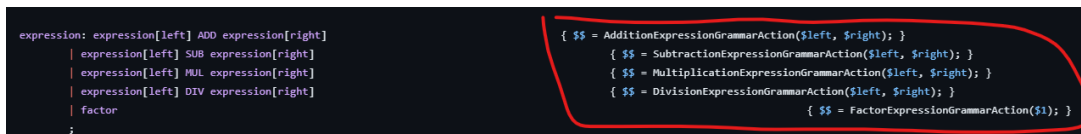
Durante el desarrollo del TP estuvimos viendo otros trabajos como inspiración, algunos bastante más viejos, y se nota enormemente el salto de calidad (del template que da la cátedra) entre TPs más viejos (2021 para atrás) y este. Lo que más se nota es que ahora el trabajo tiene estructura, hay un back, un front, y está todo explicado y con un ejemplo. Eso es un golazo. Además, tiene el *CompilerState* que simplifica un montón el manejo del estado del compilador, y el sistema de logging está buenísimo y no tenemos que hacer nada para implementarlo, funciona out-of-the-box.

El libro de Levine de “Flex & Bison” es muy bueno. Lo leímos más que nada al principio para entender la comunicación entre Flex y Bison, y nos sirvieron exactamente los capítulos que dicen en campus (“los primeros 3”).

Ahora, ¿qué mejoraríamos? Un problema al que nos enfrentamos constantemente durante el TP fue el tema de la repetición de código, cosa que se puede simplificar yendo a lenguajes como Java que tienen clases e interfaces. No sabemos si existen las herramientas en ese lenguaje para hacer lo mismo que en C con Flex & Bison, pero el comentario va más allá

del lenguaje y está más apuntado a buscar alguna alternativa para disminuir el código repetido. Una alternativa que se nos ocurre, y algo nos hubiera gustado que se mencionara en clase, sería usar *unions* y *enums* para manejar los nodos.

Otro problema que tuvimos fue que, al momento de realizar las Bison actions y generar los nodos, hicimos que por cada producción haya una función asociada distinta que se triggera, lo que hacía que el código explotara en cantidad de líneas. Esta decisión que tomamos en un principio, y que toma la gran mayoría de los grupos, se da porque, en el ejemplo que hay en el template, cada una de las acciones triggera una acción distinta, lo que lleva a uno a creer que esa es la manera de realizarlo. En nuestro caso, no podíamos concebir que fuese todo tan repetitivo, y gracias a las consultas que hicimos por mail llegamos a la conclusión que había otra alternativa usando *unions* y *enums*. Mencionar esto cuando llega el momento de hacer las actions sería de gran utilidad para muchos grupos. Hablamos de esto:



```
expression: expression[left] ADD expression[right]
          | expression[left] SUB expression[right]
          | expression[left] MUL expression[right]
          | expression[left] DIV expression[right]
          | factor
          ;

{ $$ = AdditionExpressionGrammarAction($left, $right); }
{ $$ = SubtractionExpressionGrammarAction($left, $right); }
{ $$ = MultiplicationExpressionGrammarAction($left, $right); }
{ $$ = DivisionExpressionGrammarAction($left, $right); }
{ $$ = FactorExpressionGrammarAction($1); }
```

Por último, queríamos decir que el TP nos gustó mucho y pudimos hacer gran parte de lo que nos propusimos. Creemos que, de todo lo mencionado, el hecho de hacerlo en Java puede simplificar muchísimo el código, resultando en trabajos más innovadores ya que no tendrían que pasar tanto tiempo pensando una manera para repetir menos código o quejándose de que es todo lo mismo (que es lo que nos pasó a nosotros jaja). Dicho eso, creemos también que hacerlo en C y tener que pensar en estas optimizaciones es un problema ingenieril que vale la pena tener. Aprendimos mucho más de C y de modularización gracias a este TP, por lo que el cambio a Java sería más un trade-off que una ganancia.

## 7. Referencias

- 
- [1] “Anime.js,” anime.js • JavaScript animation engine. [En línea]. Disponible: <https://animejs.com/>.
  - [2] T. Troydhanson, “uthash: C macros for hash tables and more,” GitHub. [En línea]. Disponible: <https://github.com/troydhanson/uthash>.
  - [3] Cavaliercoder, C-stringbuilder: A simple stringbuilder in C. [En línea]. Disponible: <https://github.com/cavaliercoder/c-stringbuilder>.
  - [4] “EMSCRIPTEN 3.1.42-git (DEV) documentation,” Emscripten 3.1.42-git (dev) documentation. [En línea]. Disponible: <https://emscripten.org/>.
  - [5] J. R. Levine, Flex & Bison. Beijing: O’Reilly, 2013.