

👑EmojiTeX👑

📊👤 Informe Proyecto Especial 👤📊

👉 72.39 - *Autómatas, Teoría de Lenguajes y Compiladores* 👉

👉 *Primer cuatrimestre 2024* 👉



📊 Integrantes: 📊



- 🎯 Bendayan, Alberto (Legajo: 62786) 🎯
- 🎯 Quian Blanco, Francisco (Legajo: 63006) 🎯
- 🎯 Shlamovitz, Theo (Legajo: 62087) 🎯
- 🎯 Stanfield, Theo (Legajo: 63403) 🎯



📊 Profesores: 📊



- 🎯 Arias Roig, Ana Maria 🎯
- 🎯 Golmar, Mario Agustin 🎯



Tabla de contenidos

Introducción.....	3
Modelo Computacional.....	4
Dominio.....	4
Lenguaje.....	4
Implementación.....	5
Frontend.....	5
Backend.....	6
Adicionales.....	7
Dificultades Encontradas.....	7
Cadena de texto vacía.....	7
Reestructuración para aceptar ‘\n’ y ‘ ’.....	8
Lenguaje dentro de bold/italic/underline.....	8
Lenguaje dentro listas, códigos y tablas.....	9
Visualización de emojis.....	9
Decodificación de Emojis y Backspace.....	9
Manejo Incorrecto de Lexemas y Espacios en Blanco.....	10
Aspectos a tener en cuenta.....	10
Salto de página en markdown.....	10
Sintaxis de lenguaje en bloque de código y tablas.....	10
Encadenaciones de caracteres dentro de modificadores de texto.....	11
Escritura de emojis.....	11
Escritura de ‘\n’.....	11
Limitaciones.....	11
Imposibilidad de escribir ciertos emojis en ciertas etiquetas.....	11
Modificadores de texto dentro de etiquetas.....	12
Futuras extensiones.....	12
Conclusión.....	12

Introducción

El objetivo del trabajo fue desarrollar un lenguaje que permita transformar de un lenguaje con funciones definidas por emojis, en un documento en formato LaTeX. El lenguaje permite generar un documento con un subconjunto reducido de características del lenguaje final LaTeX por cuestión de simpleza y enfoque del proyecto.

Para permitir mayor dinamismo y completitud del proyecto, por cada característica de dicho subconjunto, se definieron opciones de emojis que generan un resultado equivalente.

El compilador generará como salida un documento escrito en LaTeX, sin embargo, si se desea convertir dicho documento a formato PDF o HTML se podrá utilizar el motor compilador de LaTeX para generar dichas salidas.

La implementación satisfactoria de este lenguaje ofrecería una nueva opción para escribir documentos de forma sencilla, consistente y en texto plano equivalente a los lenguajes de markup ya conocidos al día de hoy.

De esta manera se ofrece un lenguaje introductorio a la sintaxis de LaTeX para las personas las cuales no tienen experiencia previa de una manera más simple, divertida y entretenida.

El desarrollo frontend constó en 2 distintas fases. La primera siendo el procesamiento de la entrada para la creación de los tokens y la segunda el creado de la gramática.

Para la primera parte, el analizador léxico utilizado fue Flex y con la definición de un alfabeto, fuimos capaces de ir const

Modelo Computacional

Dominio

Se desarrolló un lenguaje para convertir funciones definidas por emojis en un documento en formato LaTeX. Este lenguaje está diseñado para generar documentos que incluyan un conjunto reducido de características de LaTeX, enfocado en la simplicidad y los objetivos específicos del proyecto. Cada característica de este subconjunto tendrá opciones de emojis definidas para asegurar dinamismo y completitud en la generación del documento final.

Lenguaje

El lenguaje desarrollado ofrece las siguientes construcciones, prestaciones y funcionalidades:

- (I). Se podrá crear texto con estilo “**Bold**” mediante el encapsulamiento del texto entre emojis 🧑.
- (II). Se podrá crear texto con estilo “*Italics*” mediante el encapsulamiento del texto entre emojis 🧐.
- (III). Se podrá crear texto con estilo “Underline” mediante el encapsulamiento del texto entre emojis 📊.
- (IV). Se podrá crear texto con distintas combinaciones de los puntos I, II y III.
- (V). Se podrá crear texto con tamaño de “Título” mediante el encapsulamiento del texto entre emojis 👑.
- (V.1). El título puede aparecer una única vez en el documento y debe ser como primera línea del mismo.
- (VI). Se podrá crear texto con tamaños de “Heading 1, 2 y 3” mediante el encapsulamiento del texto entre emojis 😊, 🧑 y 🧐 respectivamente.
- (VII). Se podrán crear listas ordenadas mediante el encapsulamiento del texto entre emojis

1	2
3	4

.
- (VIII). Se podrán crear listas no ordenadas mediante el encapsulamiento del texto entre emojis ✨.
- (IX). Se podrán listar los ítems de las listas ordenadas y no ordenadas mediante el encapsulamiento del texto entre emojis 🎯.

- (X). Se podrán crear tablas mediante el encapsulamiento del texto entre emojis 📊, y cada celda se debe encerrar entre emojis 🗂️ que funcionarán como tabulador.
- (X.1). Se debe especificar la cantidad de columnas de la tabla a continuación del emoji de apertura.
- (XI). Se podrán crear bloques de código mediante el encapsulamiento del texto entre emojis 💻.
- (XI.1). Para especificar el lenguaje del bloque de código, similar a Markdown, se debe indicar el lenguaje en texto, inmediatamente después del emoji de apertura del bloque.
- (XII). Se podrán insertar imágenes mediante el encapsulamiento del texto entre emojis 📷.
- (XIII). Se podrá definir un salto de página mediante dos emojis 🐰.
- (XIV). Se podrán “escapar” emojis ya que al ser una sintaxis de emojis. De querer poner un emoji en el documento deberá ser escapado mediante el encapsulamiento del emoji 🤖.

Implementación

Frontend

El desarrollo frontend constó en 2 distintas fases. La primera siendo el procesamiento de la entrada para la creación de los tokens y la segunda el creado de la gramática.

Para la primera parte, el analizador léxico utilizado fue Flex y con la definición de un alfabeto, fuimos capaces de ir construyendo nuestra secuencia de símbolos. Definimos todos los tokens que íbamos a utilizar y creamos los contextos que supusimos necesarios, más adelante en el proyecto nos dimos cuenta de que no fueron los suficientes por lo cual tuvimos que agregar diferentes contextos para varios tokens adicionales.

Como nuestros tokens de apertura y cierre son los mismos tuvimos que pensar alguna manera para poder decidir el estado actual de la etiqueta, si está abierta o no. Luego de varias ideas, decidimos utilizar variables booleanas para representar si el siguiente token a procesar es el de apertura o el de cierre.

Luego pasamos a la segunda etapa de esta entrega parcial que es la del análisis sintáctico, en la cual utilizamos Bison. El objetivo de esto es construir el árbol de sintaxis abstracto nombrado anteriormente.

Backend

Luego del nodo principal, llamado programa, creamos el nodo de título. Ya que el documento puede o no tener título, y si es que tiene debe estar al principio, decidimos respetar esto para facilitar el creado del documento posteriormente. Este nodo, tiene tres posibilidades distintas, las cuales son título, sin título y título solitario para cumplir con todas las posibilidades distintas.

En caso de que no sea un título solitario, el documento incluye lo que llamamos etiquetas, similares a las que se encuentran en HTML. Utilizamos este nombre pues se asimilaran mucho a las utilizadas en HTML por ejemplo para distintos encabezados, inclusión de imagen o de bloque de código, o de listas ordenadas y no ordenadas. Para el nodo de las etiquetas, tenemos todos los tipos de etiquetas posibles incluyendo la etiqueta de texto plano.

Las etiquetas las podemos distinguir entre dos grupos, las básicas y las compuestas. Aunque esta separación no es tan notoria en un principio, podemos notar como hay ciertas etiquetas que, dentro de ellas, tienen simplemente texto y otras que llevan más cosas dentro de ellas. Etiquetas como las del encabezado, o la etiqueta para escapar emojis, dentro suyo simplemente llevan texto, lo cual concluye que su nodo simplemente tenga un puntero al nodo del texto. En cambio, otras etiquetas como las listas o las tablas que contienen, en el caso de las listas ítems y en el caso de las tablas celdas, su nodo tendrá un puntero a una lista enlazada de ítems celdas.

Por último, los nodos de los modificadores de texto que pertenecen a ambos grupos de etiquetas simultáneamente. Debido a que realizamos las combinaciones de todos los modificadores, en caso de que sea simplemente un texto en negrita, itálica o subrayado, pertenece a una etiqueta simple, pero cuando se realizan las combinaciones ya pertenecen a las etiquetas compuestas. Para esto se crearon nodos para todas las combinaciones lo cual resultó ser un trabajo muy tedioso, pero eficaz al fin.

Nos pareció interesante esta separación de las etiquetas pues hacía más fácil ver que etiquetas iban a tener simplemente un puntero apuntando a un lista enlazada de caracteres, y que otras etiquetas tendrían que tener sus propios nodos y punteros hacia ellos.

Adicionales

Una función adicional que contiene el trabajo desarrollado es que se puede obtener como resultado, aparte de un documento LaTeX, un documento Markdown.

Para obtener la salida en Markdown se debe compilar con el indicador “`--md`”, lo cual utilizará el archivo “`generatorMD.c`”, el cual se encuentra dentro de la carpeta “`extensiones`”, en vez del archivo “`generator.c`” que es el archivo estándar utilizado. Es decir, al compilar normalmente el proyecto se obtendrá un documento en LaTeX, en cambio al utilizar “`--md`”, se obtendrá un documento en Markdown.

Dificultades Encontradas

Durante el desarrollo del proyecto nos hemos cruzado con distintos problemas los cuales fuimos resolviendo en equipo y en otras ocasiones tuvimos que pedir ayuda a la cátedra. A continuación detallamos las distintas dificultades encontradas a la hora de realizar el trabajo y cómo se solucionaron.

Cadena de texto vacía

El primer inconveniente surgió durante la primera entrega del proyecto, en la cual la gran mayoría de las pruebas fallaban y no lográbamos identificar la causa de estos errores. Debido a la proximidad de la fecha de entrega, nos vimos obligados a presentar un trabajo que no funcionaba al 100%. Sin embargo, a pesar de no haber entregado en condiciones óptimas, continuamos revisando el código con el objetivo de encontrar el error. Finalmente, llegamos a la conclusión de que el fallo se debía a no haber considerado el caso de aceptar una cadena de texto vacía al finalizar una sentencia de reglas con emojis.

Después de identificar y corregir el error, todos los tests pasaron con éxito y finalmente el chequeo de sintaxis funcionó de manera óptima. Este proceso fue crucial para asegurar la funcionalidad y la robustez del código. Los errores, aunque indeseables en la vida de un programador, pueden ser oportunidades valiosas para profundizar en la comprensión del código y mejorar las habilidades de resolución de problemas.

En este caso particular, el error nos permitió comprender mejor la estructura y el comportamiento del sistema. La necesidad de considerar todos los posibles casos, se hizo evidente. Al no haber contemplado la aceptación de una cadena de texto vacía al finalizar una

sentencia de reglas con emojis, enfrentamos dificultades significativas que nos impidieron avanzar en el proyecto.

Este incidente subraya la importancia de una revisión exhaustiva y meticulosa del código, así como la consideración de todos los escenarios posibles durante el proceso de desarrollo. A través de este desafío, no solo logramos solucionar el problema específico, sino que también adquirimos una perspectiva más amplia sobre la importancia de la atención al detalle y la previsión en el desarrollo de software.

Reestructuración para aceptar ‘\n’ y ‘ ’

Una vez arreglados los tests, decidimos escribir nuevos tests ya que no estábamos comprobando todos los casos que queríamos comprobar para nuestro compilador. Al agregar un test que incluye una cadena de texto incluyendo caracteres como ‘\n’, notamos que no los estábamos guardando en el flex. Esto supuso un gran problema pues los espacios tampoco los estamos guardando sino que los agregábamos manualmente, es decir, cada vez que se escribía una palabra se escribía la misma con un espacio a la derecha. Esto llevaba a que la última palabra de la oración o del párrafo incluya un espacio vacío al final, un pequeño error que podía ser ignorable.

Luego de haber leído mucha documentación de flex, encontramos que el builtin ‘[:string:]’, el cual estábamos ignorando, incluye muchos caracteres, dentro de ellos caracteres como ‘\n’ y ‘\t’ los cuales no queremos ignorar. Para solucionar esto, decidimos no ignorar estos caracteres sino aceptarlos y guardarlos, así de esta manera poder imprimirlos directamente y no tener que agregar manualmente los espacios, lo cual soluciona el problema del espacio de más que se agregaba cuando se terminaba un párrafo u oración.

Lenguaje dentro de bold/italic/underline

Una vez que logramos aceptar los distintos caracteres especiales, surgió un problema inesperado. Como aceptamos el carácter ‘\n’ sin importar dentro de qué etiqueta estemos, también se acepta dentro de las etiquetas para modificación de texto (bold/italic/underline). Como en LaTeX esto se comporta de manera extraña, fallando en algunos motores, decidimos que sería buena idea emitirlos. Esto resultó ser un problema pues la manera de solucionarlo fue crear distintos contextos para cada combinación de las etiquetas de modificación de texto y de esta manera, dentro de estos contextos ignorar el ‘\n’. Esto resultó


ser algo positivo pues obtuvimos una manera de distinguir las distintas etiquetas y cómo modificar los lenguajes que cada uno aceptan, también permitiendo emitir emojis dentro de estas etiquetas.

Lenguaje dentro listas, códigos y tablas

De igual manera que mencionamos anteriormente, tuvimos que cambiar el flex para que las listas, códigos y tablas, tengan su propio contexto y de esta manera manejar las distintas etiquetas y sus lenguajes independientemente de las otras etiquetas. Fue importante crear los distintos contextos porque los lenguajes dentro de estas etiquetas son distintos, es decir, los caracteres que se aceptan o que se ignoran en cada etiqueta varían dependiendo de en qué etiqueta te encuentras. Para dar un ejemplo, dentro de las tablas no se puede incluir un ‘\n’ pues rompería la etiqueta de la tabla, pero dentro de un bloque de código sí. También los emojis que son tratados como texto o como token varían dependiendo del contexto.

Este problema fue importante de solucionar pues la funcionalidad de cada etiqueta depende del lenguaje que usa, es decir, de los caracteres permitidos y no permitidos, por ejemplo, un bloque de código formateado debe incluir los distintos ‘\n’ y ‘\t’ para respetar el formato correcto.

Visualización de emojis

El otro gran problema que hemos tenido fue finalizando el proyecto donde, una de las funciones que la cátedra provee, no podía imprimir bien los emojis. En lugar de mostrar el emoji correspondiente, visualizábamos el carácter . El error se encontraba en el manejo de caracteres Unicode, específicamente en cómo nuestra implementación gestiona emojis y otros caracteres multibyte. Aquí, el error se manifiesta en dos áreas críticas:

Decodificación de Emojis y Backspace

Cuando se utiliza la tecla backspace sobre un emoji, aunque visualmente el emoji se mantiene, se elimina una secuencia subyacente de 9 bytes. Esto se debe a que muchos glifos de Unicode se componen de múltiples secuencias de bytes. Ejemplo: "á" se codifica como una "a" y un tilde, que son dos caracteres separados.

Manejo Incorrecto de Lexemas y Espacios en Blanco

En nuestro código, había un manejo incorrecto que provocaba la adición de espacios en blanco no deseados, complicando la correcta identificación de lexemas y whitespace.

Para poder resolver este problema, tuvimos que corregir bastantes cosas, comenzando con la adición de espacios en blanco para que los caracteres puedan ser tratados correctamente y de esta manera tratar individualmente cada parte del emoji, y a la hora de mostrarlo por pantalla juntar todas estas partes para poder mostrar el emoji completo.

Finalmente, pudimos solucionar todos los problemas encontrados pudiendo “pasar por arriba” Flex para que este pueda manejar Unicode y que la implementación de nuestro proyecto sea exitosa.

Aspectos a tener en cuenta

Salto de página en markdown

El salto de página en markdown no existe como tal, sino que se suele incluir una etiqueta `<div>` de HTML con la clase para que se produzca el salto de página, debido a que markdown admite sintaxis HTML. Esto es lo mismo que realiza el compilador pero hay un caso en el cual no funciona. Si el visualizador que se está usando para ver el código de markdown tiene separación entre distintas páginas, se verá adecuadamente. Pero, en caso de que no tenga distintas páginas, sino que sea simplemente una página que se extiende infinitamente, no se podrá apreciar un salto de página debido a que no hay páginas.

Sintaxis de lenguaje en bloque de código y tablas

Para describir el lenguaje de programación en el cual se programó un código al incluir este en un bloque de código se debe incluir inmediatamente el nombre del lenguaje luego de emoji de la computadora sin dejar espacios. De dejar algún espacio, insertar un ‘\n’ o insertar algún carácter previo al texto con el lenguaje, no funcionará.

Similar al caso de los bloques de código, para el caso de las tablas, luego de comenzar la sentencia de tablas con un emoji de apertura, se debe especificar la cantidad de columnas que se desea tener en la tabla, previo a especificar los delimitadores de celdas (junto con sus contenidos). A diferencia del bloque de código, no es requisito que dicha cantidad de

columnas se encuentre inmediatamente después del emoji de apertura, es decir, pueden haber espacios en blanco. Esto se da porque dentro del contexto de las tablas, al momento de su procesamiento, los espacios en blanco son ignorados, por lo que no pueden existir saltos de línea ni múltiples espacios en blanco en la tabla. Esto se diferencia del contexto del bloque de código donde, como queremos mantener la estructura del código, resultaba una necesidad explícita aceptar los espacios en blanco (incluyendo saltos de línea y tabulaciones) como parte del texto a incluir dentro del resultado final de la compilación.

Encadenaciones de caracteres dentro de modificadores de texto

Otro aspecto importante que no afecta la funcionalidades del compilador es el hecho de que no admita encadenamiento de palabras dentro de los modificadores. Para dar un ejemplo, el compilador no admite “👤 Bold 🍷 Italics 🍷 👤” pues hay una encadenación de texto y luego italics. La razón por la cual esto no quita funcionalidad al compilador es porque esta misma oración puede ser escrita como “👤 Bold 👤👤 🍷 Italics 🍷 👤” resultando en el texto esperado.

Escritura de emojis

Para poder escribir un emoji sin estar dentro de ninguna etiqueta se deberán escapar con el emoji del policía. En caso de estar dentro de etiquetas como tablas, listas o bloques de código se puede escribir directamente el emoji.

Escritura de ‘\n’

Para poder separar dos párrafos en LaTeX o en Markdown, es decir, forzar un salto de línea, se debe dejar una línea vacía entre ambos. Es decir, dos saltos de líneas en texto plano, para que funcione correctamente en nuestro compilador se debe realizar lo mismo.

Limitaciones

Imposibilidad de escribir ciertos emojis en ciertas etiquetas

A pesar de haber incluido un contexto para poder escapar caracteres, dentro de ciertas etiquetas no se pueden escapar algunos emojis.

- El emoji 👮 no puede ser utilizado sin estar dentro de un etiqueta

- El emoji 🎯 no puede ser usado dentro de una lista
- El emoji 🦄 no puede ser usado dentro de una tabla

Modificadores de texto dentro de etiquetas

Otra limitación de nuestro compilador es la imposibilidad de poder escribir modificadores de texto, como bold/italic/underline dentro de varias etiquetas. En LaTeX, se pueden utilizar los modificadores de textos en listas, tablas y otras etiquetas. Nuestro compilador no acepta la utilización de estos modificadores dentro de otras etiquetas, simplemente como etiquetas independientes y sus combinaciones entre ellos.

Futuras extensiones

La primera funcionalidad a desarrollar sería implementar la transformación mediante emojis para todo el lenguaje LaTeX. Consideramos que las funcionalidades ya aplicadas en el proyecto son más que suficientes para poder crear un lenguaje interactivo introductorio a LaTeX pero, a su vez, consideramos interesante la posibilidad de poder hacer el lenguaje de emojis más completo.

Otra extensión posible al proyecto en un futuro es hacer una interfaz gráfica donde un usuario en vez de tener que escribir código, pueda hacerlo aún más interactivo mediante la utilización del arrastre de bloques con la utilización de emojis haciendo aún más simple, divertida y entretenida la introducción al lenguaje LaTeX.

Finalmente, otra posible extensión del proyecto sería implementar la conversión a PDF y/o HTML de la salida del proyecto dentro del mismo sin la necesidad de tener que utilizar el motor de LaTeX para la conversión.

Conclusión

El trabajo nos obligó a investigar y profundizar nuestros conocimientos. Fue un trabajo con muchos obstáculos, pero trabajando en equipo y, con ayuda de la cátedra, consideramos que pudimos superarlos. A pesar de que estos obstáculos fueron frustrantes en sus momentos, al terminar el trabajo podemos decir que son estos mismos los que nos forzaron a adquirir nuevos conocimientos.

Por último, lo más importante es que como equipo, disfrutamos el trabajo ya que a pesar de que por ciertos momentos fue frustrante, pudimos resolver los obstáculos. Consideramos que como grupo trabajamos bien entre los 4, ya que por no decir todo, la gran mayoría del trabajo, lo realizamos mediante la plataforma “Discord” en llamada, así pudiendo discutir lo que se iba realizando el trabajo los 4 juntos mediante la utilización de la extensión “Live share”, lo que permite escribir código de a más de a 1 persona a la vez en el mismo file.