72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Felipe Hiba, 61219
fhiba@itba.edu.ar
Pedro Curti, 61616
pcurti@itba.edu.ar

20 de Junio de 2024

Introducción

Nuestro equipo decidió desarrollar un compilador de markdown que ofrece las funcionalidades básicas de este, es decir no tenemos la sintaxis extendida. Además, restringimos la multiplicidad de opciones que ofrece MarkDown por defecto para simplificar su uso. A este también le agregamos nuestra propia sintaxis donde el usuario puede agregar modificadores de estilo.. La finalidad de este proyecto es poder ofrecer más opciones de estilo de lo que hace markdown, a través de un formato simple manteniendo la idea original, que es que sea además de formateable, legible a simple vista.

Modelo Computacional

El compilador acepta todo tipo de modificadores que acepta MarkDown sin sus extensiones. Esto incluye:

Funcion	Símbolo
Header	#
Bold	*
Italic	_
BlockQuote	>
`CodeBlock`	`
Link Interno	[headerEjemplo](#link-al-header)
Link Externo	[link a la pagina](https://www.pagina.com)
Ordered List	1.
Unordered List	-
Comentario	[//]: # “”

El nivel del header depende de cuantos # se utilizan, entre 1 y 6. Cuantos más # más chico es el header y luego del sexto # se utiliza el nivel 6. En cuanto a las listas, estas llegan hasta un tercer nivel de indentación y luego, como los headers, si se excede del tercer nivel entonces se toma el máximo. Para los links internos verificamos que el header al que refiere efectivamente exista, si este no existe, el programa no se ejecuta y termina con un mensaje de error donde loguea porque es que falló el programa. Todos los formatos son combinables en cualquier orden pero requieren que se respete el formato necesario para cada uno. Es decir, para los headers debe haber un espacio luego del # y debe estar al principio de la linea, luego de poner un * o un _ no puede haber un espacio si se quiere que se ejecute el bold o el italic. En cuanto a los links, utilizamos el mismo formato de markdown donde entre los () va el link al que se refiere y luego entre los [] va el nombre que se va a mostrar en el archivo que cuando sea clickeado nos va a llevar al archivo referido.

Luego con respecto a lo que nosotros le agregamos, el formato del comentario se ve modificado para que nosotros podamos interpretar con el compilador que ese no és un comentario cualquiera, sino que es un comentario que va a agregar estilos al ítem que esté abajo.

Funcion	Símbolo
Comentario de estilo	[//]: # “# “

Los comentarios que no utilizan ese # dentro de las “” no son renderizados en el archivo de salida y tampoco se tienen en cuenta para la estilización del archivo, de esta manera permitimos que se pueda comentar el código por si el usuario quiere, de la misma manera que se puede en cualquier lenguaje de programación.

Nuestros modificadores de estilo utilizan el siguiente formato:

Funcion	**Símbolo**
Underline	u: true | false
Underline Color	uc: color_name | #ffffff
Background Color	bg: color_name | #ffffff
Font Color	fc: color_name | #ffffff
Font Size	fs: [0-9]*px
Font Family	ff: font_family
Position	position: left | center | right

Con respecto a la gramática que propusimos en una primera instancia, mantuvimos todo lo que se planteó inicialmente.

Implementación

FrontEnd

Para comenzar, exploramos las distintas sintaxis que ofrece Markdown para generar los distintos bloques y formatos que ofrece de entrada, ya que hay varias maneras de llegar al mismo resultado. Un ejemplo de esto es cómo generar los Headers. Podría hacerlo agregando un '#' al principio de la línea, o también, otra opción que es utilizada, en la siguiente línea poner '===' o '---'. Este tipo de decisiones, donde nos decantamos por un formato u otro, fueron tomadas basándonos en nuestra experiencia y preferencias sobre el formato MarkDown original, y porque creemos que a veces, esta variedad de reglas puede ser confusa o molesta, como es el caso de generar texto **bold** o *italic*, donde cambia dependiendo la cantidad de caracteres que se use, es decir, esto es **bold** y es *italic*. Lo mismo aplica con los "_". Esto parecía confuso, por eso nos decantamos por el uso de una sola regla para cada función ofrecida.

Luego buscamos un sistema para que nuestras sintaxis, que solo busca expandir la original, no colisione con la misma, o prive el uso de esta. Encontramos que hay comentarios en Markdown, como en otros lenguajes de programación, entonces simplemente, nuestra sintaxis iría en los comentarios de Markdown, pero marcados de manera especial, para diferenciarlos de comentarios normales.

Después de definir estas reglas propias, buscamos cómo aplicarlas a flex. Para comenzar, la mayoría de funciones de Markdown, se aplican al comienzo de las líneas, sean headers, listas, comentarios normales, nuestros comentarios con estilo o blockquotes. El problema de markdown es que al ser un lenguaje para texto normal, a diferencia de un lenguaje de programación, este es bastante más laxo, ya que un símbolo especial depende de donde se ubica, puede ser palabra reservada, o un símbolo cualquiera. En cambio un lenguaje de programación como C, son más estructurados y por ejemplo, un ";" siempre implica el cierre de una expresión. Por lo tanto, en flex utilizamos las expresiones regulares para garantizar que si se usa un carácter especial, tiene que usar el formato especificado para que se aplique el estilo. Es decir que si se utiliza un # para generar un Header, tiene que haber un espacio después del # y tiene que estar al principio de la línea. De esta manera garantizamos que el usuario efectivamente quería usar el # para generar un Header. Esto aplica a todos los tags que se utilizan al comienzo.

Luego tuvimos que crear las reglas para el inline styling. Esto tiene su complejidad, ya que se pueden aplicar unos sobre otros, y en cualquier orden. Además que no debe aplicarse a menos que, los símbolos para estos bloques, están en contacto con caracteres distintos de espacios. El styling en ese sentido fue más sencillo, porque podemos usar un contexto aparte de flex para crearlos, ya que dentro de estos comentarios no debiera haber bloques que no sean del estilo que ofrecemos nosotros. De nuevo, podemos ver la ventaja que tiene a la hora de pensar estas reglas, lenguajes más estructurados, que permiten sacarle provecho real a los contextos de flex.

Desde Bison, el trabajo estuvo centrado en conseguir el mejor output AST posible, para que luego, el backend no tenga demasiadas dificultades para construir el archivo final. Esto cobra especial importancia a la hora de armar las listas, ya que pueden ser anidadas. Es sencillo a simple vista, pero tiene sus complejidades si quiere evitarse tener conflictos **shift/reduce** o **reduce/reduce**. Aunque en la práctica, los primeros son permitidos, puesto que Bison posee como política decantar por *shift* en vez de *reduce* si es que existiera un conflicto. No aplica para el Trabajo Práctico de TLA.

BackEnd

Dentro del backend, tenemos un primer chequeo que es domain-specific donde nos aseguramos que los links internos que se van a generar estén refiriendo a un header que también existe dentro de ese mismo archivo. Esto lo hacemos recorriendo todos los nodos que generamos dentro del programa y entrando hasta el bloque de cada uno para revisar si hay un link o un header ahí. Con los links la búsqueda debe ser más exhaustiva ya que pueden haber links integrados dentro de cualquier parte de texto, sin importar el bloque, en cambio para los headers es más simple, ya que solo pueden estar dentro de un header block. Una vez que encontramos cualquiera de estos, lo guardamos en una lista hasta que terminemos de revisar todos los bloques.

Al finalizar la búsqueda de headers y links, hacemos el chequeo. Este chequeo se basa en ciclar la lista de headers por cada link que haya dentro de la lista de links. No podemos ir borrando los headers que ya encontramos debido a que un mismo header puede estar referido más de una vez. Si el link encuentra el header al que refiere entonces continua la busqueda, pero con tan solo encontrar un link que haya ciclado la lista completa de headers sin encontrar su par, detenemos el proceso y generamos un exit con código de error avisando al usuario que su archivo de markdown está mal formateado.

Luego de chequear que los links existan, pasamos a la generación del archivo de output. Dentro del generator abrimos un archivo output.html donde vamos a ir cargando todos los tags generados por el archivo de markdown de input.

Comenzamos con un prólogo donde se insertan todos los headers que lleva el archivo de HTML para formatearlo de manera correcta, abrimos los tags de body y html y luego pasamos a la generación del programa.

En generate program vamos generando recursivamente los tags desde adentro hacia afuera para poder respetar de manera correcta el orden de los tags y también el orden de los modificadores que se le pusieron a cada uno de los ítems. Primero generamos el bloque dependiendo del tipo general que sea, por más que la generación sea de adentro hacia afuera, la escritura al output es secuencial, por lo que debemos ir abriendo los tags a medida que vamos entrando en cada tipo de bloque. Dentro de cada bloque nos fijamos si hay algún estilo que se le deba agregar dentro del tag y si lo hay lo agregamos antes de seguir entrando en la estructura del bloque. Finalmente llegamos al final de la estructura cuando llegamos al texto plano y no hay nada más a lo que podamos acceder dentro de la estructura, y es ahí cuando empezamos a salir de las funciones cerrando los tags respectivamente.

Finalmente, cuando ya todos los bloques de código fueron generados y generate program terminó, vamos a un epilogue donde se cierran todas los tags generales, como body y html que fueron abiertos en el prologue, y cerramos el archivo output.html que queda en la carpeta general del proyecto.

Dificultades Encontradas

Cuando desarrollamos el front no nos encontramos con ningún inconveniente grande, más allá de lo explayado en la sección de frontend, que son dificultades inherentes a la elección del proyecto. Aunque hay una curva de aprendizaje del uso de las herramientas, donde a base de prueba y error uno encuentra mejores estrategias para crear reglas sintácticas y semánticas. Creíamos que había quedado todo solucionado de manera correcta pero lo que no habíamos previsto bien había sido el manejo de las listas en el back. Ya que las listas pueden ser ordenadas y desordenadas, y estas mismas pueden estar anidadas. Tuvimos que revisar la manera en la que guardábamos las listas dentro de los bloques respectivos para luego en el generate poder manejar de manera correcta la creación de listas. Lo que nos ocurría al principio era que las listas eran generadas pero nos dábamos cuenta de que se generaban como listas individuales, entonces en vez de tener

1.
2.
3.

lo que nos ocurría era que nosotros generabamos

1.
1.
1.

debido a que los tags de las listas se cerraban luego de cada bloque porque estos no estaban asociados entre sí. Por lo que tuvimos que repensar la lógica detrás de cómo nos guardabamos las listas desde el front y para evitar el backtracking de tener que contar la cantidad de indentaciones. Dependiendo de la cantidad de indentaciones, desde flex llamamos a una LexemeAction distinta para resolver el problema y ahí guardarnos este valor. Luego desde el back utilizamos una seguidilla de lógicas que implicaba guardarnos si el nodo anterior era una lista, cuál fue su profundidad y un array donde nos guardabamos que tipo de lista hay en qué profundidad para luego saber como cerrar o abrir los tags del próximo bloque. Esto no es menor, puesto que depende cuando se abra o cierre una lista, el HTML que luego se ve, cambia esencialmente.

Futuras Extensiones

Las posibles futuras extensiones que vemos posibles con nuestro proyecto son agregar más modificadores de estilo dentro de los que nosotros ya ofrecemos para poder darle más personalización al usuario al momento de generar sus archivos html. Por otro lado, también se podría extender la cantidad de funciones de markdown que abarcamos para permitir que se generen funciones en latex o tablas complejas como se puede hacer en versiones más actualizadas de markdown que implementan la sintaxis extendida.

Conclusiones

Durante el desarrollo del proyecto logramos aprender nuevas tecnologías como el uso de flex y bison para generar nuestro compilador. Descubrimos que la mejor manera para encarar los proyectos donde la complejidad se eleva muy rápido es empezar por los casos más básicos y luego crecer de a poco buscando las excepciones en vez de tratar de generar a la primera el resultado que abarque todas las reglas posibles. Esto nos sucedió bastante cuando estábamos desarrollando la parte de las listas ya que intentábamos hacer una lógica que abarque todas las posibilidades de las listas de una pero esto se complicaba y terminamos con estructuras complejisimas y sin sentido cuando en realidad el producto final fue muchisimo mas simple y reducido por el simple hecho de empezar de a poco. Por otro lado, el uso de los tests también nos ayudó a entender por donde estábamos fallando en la producción del archivo final, es por esto que tener una buena cantidad de tests simples que sean explicativos y que testeen casi unitariamente es una parte clave de este proyecto. Además la herramienta de logueo ofrecida por la cátedra fue clave, ya que permite detectar en qué función exacta había problemas, y permite recorrer mentalmente el código sin tener que usar herramientas complejas de debugeo de C, acelerando el proceso y manteniendo un nivel de abstracción que permite no perderse demasiado en las líneas de código escritas. Pudimos ver problemas como en la generación de los estilos desde los tests ya que de a poco íbamos probando cada uno y no todos juntos de una.

Bibliografía

Para la sintaxis:

- https://daringfireball.net/projects/markdown/
- https://markdown.es/sintaxis-markdown/
- https://www.markdownguide.org/