



ITESO

Universidad Jesuita
de Guadalajara

Lenguajes Formales

Proyecto Final

Text Editor with Markdown

Integrantes

Julián de Jesús López López

María del Carmen Martínez Nuño

Mariana Sierra Vega

Profesor - Luis Eduardo Pérez Bernal

Descripción del problema a resolver

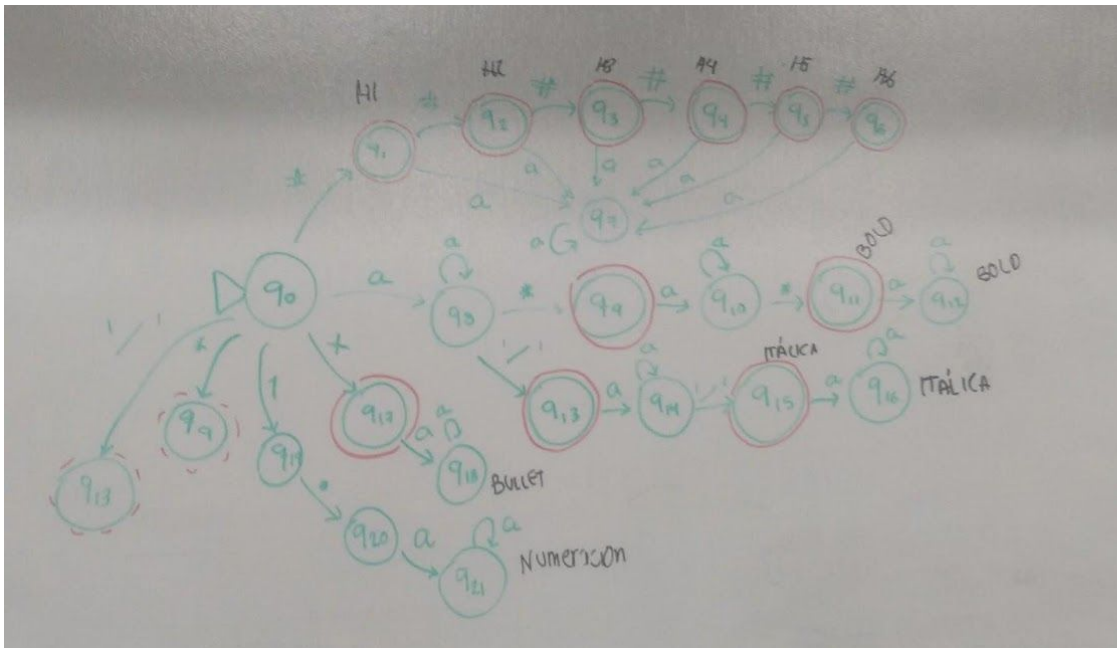
Tener aplicaciones para editar texto es indispensable, como humanidad necesitamos escribir y poder plasmar nuestros pensamientos en papel o en este caso en la computadora. Dentro de las aplicaciones actuales de tipo editor de texto encontrar las opciones para modificar el estilo, tamaño, tipo de letra entre otras características es difícil, consume tiempo e implica despegar tus manos del teclado lo que impide el flujo creativo de las palabras.

Justificación

Se propone una aplicación de editor de texto simple y minimalista que incluya etiquetas fáciles de usar, tipo *markdown*, para poder editar el estilo del texto. Para esta solución se utilizarán Autómatas Finitos Deterministas para identificar las diferentes etiquetas disponibles ya que los pasos a seguir están perfectamente bien definidos.

Análisis y diseño del algoritmo

Para llegar a la solución planteada se definió el siguiente autómata finito determinista:



El estado inicial de este autómata tiene tantos caminos como etiquetas. Éste consta de 23 estados, por los cuales se analizará cuál es la etiqueta que se va a aplicar al texto.

Implementación

Frontend

La implementación del código consta de varias partes como la interfaz gráfica (implementada con JavaFX), las clases para el autómata (definido como backend), las clases que se encargan de conectar la parte gráfica con la parte funcional (los controladores), los archivos de vistas para los estilos que se aplicarán (archivos css), y clases con funciones genéricas que nos ayudan a realizar funciones básicas como la creación de ventanas y la comunicación entre éstas.

La parte gráfica cuenta con dos ventanas principales, las cuales fueron creadas con archivos “.fxml”. Una de las vistas es el editor de texto, el cual es un *TextArea* que almacena el contenido con las etiquetas a aplicar. La segunda pantalla se llama *Preview* la cual permite al usuario previsualizar el texto con las etiquetas ya aplicadas sobre éste.

La manera para manipular el texto fue la siguiente:

El texto es convertido por renglones, esto quiere decir que, se toma cada renglón del *textArea* y se envía al autómata para analizar la etiqueta a aplicar. Éste regresa una cadena dividida por “|” en donde el primer elemento es la cadena sin las etiquetas, el segundo elemento son los índices de principio y fin en los que se aplicará el estilo. Por último se envía una etiqueta con el estilo a aplicar. Ésto se puede observar en el siguiente ejemplo:

Se envía al autómata: “Este es un *texto* con negritas” y éste regresa: “Este es un texto con negritas|11,15|Bold”. Con esta información se genera un objeto de tipo *Text* con el contenido y al cual se le aplicará el estilo. Cada objeto es almacenado en un arreglo, éste es enviado de la vista del editor de texto a la ventana para mostrar el texto con los estilos. El controlador de la ventana *Preview* toma cada elemento del arreglo y lo muestra.

Backend

La implementación del backend o el autómata es constituido por la clase *State* que contiene la posición, nombre, características y transiciones del estado.

```
public class State {
    String name;
    int position;
    boolean final_state;
    boolean initial_state;
    boolean check_num;
    boolean signed;
    HashMap <Character, Integer> transition;
    String css = null;
}
```

Las transiciones se guardan directamente en un *HashMap* utilizando como llave el símbolo que se tiene que consumir para pasar al siguiente estado.

Cada estado guarda el css al que pertenece, al igual que sus características básicas, si es final, inicial o signado.

Un estado se considera final cuando la cadena es aceptada por alguna etiqueta del formato markdown; es decir es necesario aplicar algún tipo de formato css especial. Un estado se considera signado cuando se tiene que eliminar de la cadena analizada el símbolo con el que se llegó a el estado signado. Estas dos características se consideran indispensables para el análisis de las cadenas y transformación al estilo determinado por el markdown.

Gracias a los estados se construye la clase *AFD*, que cuenta con un arreglo de estados, el número total de estados, símbolos y un arreglo de símbolos.

```
public class AFD {
    int total_states;
    State states[];
    int total_symbols;
    char symbols[];
    char default_key;
    private int count_states;
```

Estas dos clases base tienen las funciones necesarias para crear cualquier Autómata Finito Determinista, obtener el css y analizar cadenas. Esta generalización nos permite poder crear otra clase en donde creamos un AFD y le asignamos las transiciones, estados y características del autómata que diseñamos para generar el markdown; esta clase llamada *markdownAFD* la mandamos llamar cuando el usuario ha ingresado texto en la pantalla principal y quiere general el preview.

```
public class MarkdownAFD {

    public static void mainAFD(AFD markdown){
        markdown.createAFD(23, 7);
        char[] symbols = {'a','#','*','/','+','1','.'};
        markdown.setSymbols(symbols);
        State auxiliar_state;
        for(int i = 0; i < 23;i++){
            auxiliar_state = new State(Integer.toString(i), false, false, i);
            markdown.addState(auxiliar_state);
        }

        markdown.states[0].initial_state = true;

        markdown.states[1].final_state = true;
        markdown.states[2].final_state = true;
        markdown.states[3].final_state = true;
        markdown.states[4].final_state = true;
        markdown.states[5].final_state = true;
        markdown.states[6].final_state = true;
        markdown.states[7].final_state = true;
        markdown.states[11].final_state = true;
        markdown.states[12].final_state = true;
        markdown.states[15].final_state = true;
        markdown.states[16].final_state = true;
        markdown.states[18].final_state = true;

        markdown.states[0].addTransition('#', 1);
        markdown.states[0].addTransition('a', 8);
        markdown.states[0].addTransition('+', 17);
        markdown.states[0].addTransition('1', 19);
```

Ahora, cuando una cadena se debe analizar primero se procesa y se determina si la cadena es aceptada o no por el autómata, utilizando el formato de salida explicado anteriormente.

Para conseguir esta salida primero analizamos los índices de la cadena en donde se aplicará el formato considerando que los caracteres especiales, se han eliminado.

```
public int[] IntProcess(String string){
    int index[] = new int[2];
    char test[] = string.toCharArray();
    int current_state = initialState();
    boolean firstIndex = false;

    if(!states[getFinalStatePosition(string)].final_state){
        index[0] = 0;
        index[1] = test.length-1;
        return index;
    }

    for(int i = 0; i < test.length; i++){
        if(states[current_state].check_num){
            if(Character.isDigit(test[i])){
                current_state = states[current_state].getTransition('1', default_key);
            }else{
                current_state = states[current_state].getTransition(test[i], default_key);
            }
        }else{
            current_state = states[current_state].getTransition(test[i], default_key);
        }
    }

    if(states[current_state].signed){
        if(!firstIndex) {
            index[0] = i;
            firstIndex = true;
        }
        else{
            index[1] = i-2;
        }
    }
}

return index;
```

Después hacemos el mismo proceso pero esta vez obtenemos la cadena procesada; sin los caracteres especiales ('#', '+', etc).

```
public String StringProcess(String string){
    char test[] = string.toCharArray();
    String finalString = "";
    int current_state = initialState();

    if(!states[getFinalStatePosition(string)].final_state){
        return string;
    }

    for(int i = 0; i < test.length; i++){
        if(states[current_state].check_num){
            if(Character.isDigit(test[i])){
                current_state = states[current_state].getTransition('1', default_key);
            }else{
                current_state = states[current_state].getTransition(test[i], default_key);
            }
        }else{
            current_state = states[current_state].getTransition(test[i], default_key);
        }
        if(!states[current_state].signed){
            finalString += test[i];
        }
    }
    return finalString;
}
```

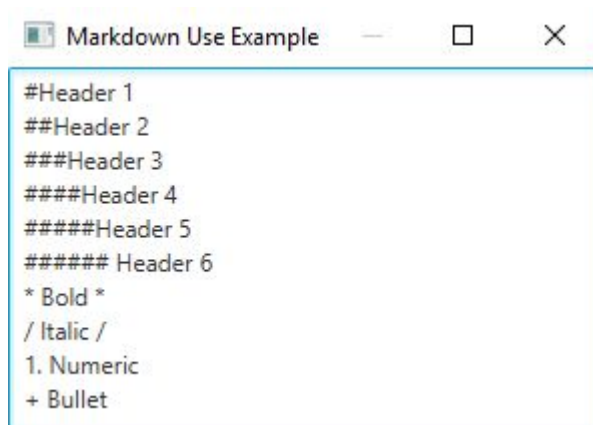

Finalmente obtenemos el css del estado final, si el estado final no tiene un css determinado; es decir, no se le tiene que aplicar una etiqueta del markdown se le deja un estilo predeterminado.

```
public String getCSS(String string){
    String last_css = null;
    char test[] = string.toCharArray();
    int current_state = initialState();
    for(int i = 0; i < test.length; i++){
        if(states[current_state].check_num){
            if(Character.isDigit(test[i])){
                current_state = states[current_state].getTransition('1', default_key);
            }else{
                current_state = states[current_state].getTransition(test[i], default_key);
            }
        }else{
            current_state = states[current_state].getTransition(test[i], default_key);
        }
        if(current_state == -1) return "No style";
        if(states[current_state].css != null){
            last_css = states[current_state].css;
        }
    }
    return last_css != null ? last_css:"Normal";
}
```

Así es como se construye la salida del autómata. Que después el procesador analiza para poder aplicar el formato final.

Ejecución

A continuación se muestra la nomenclatura para el uso del formato Markdown en la aplicación.



Como se puede observar se tienen 4 tipos diferentes de marcas:

1. “#” Por cada # se va un nivel abajo de Título.
2. *Palabra* Lo que se encuentre entre los símbolos “*” se marcará como formato **negritas**.
3. /Palabra/ Lo que se encuentre entre los símbolos “/” se marcará en formato *itálica*.
4. “+” Texto renglón; Al usar un símbolo de “+” el renglón en cuestión se hará en formato bullet “>>”.

5. Al usar un número y enseguida un punto “.” se marcará de una manera parecida al bullet pero, en vez de tener los símbolos “>>” se tendrá los números con formato de lista.

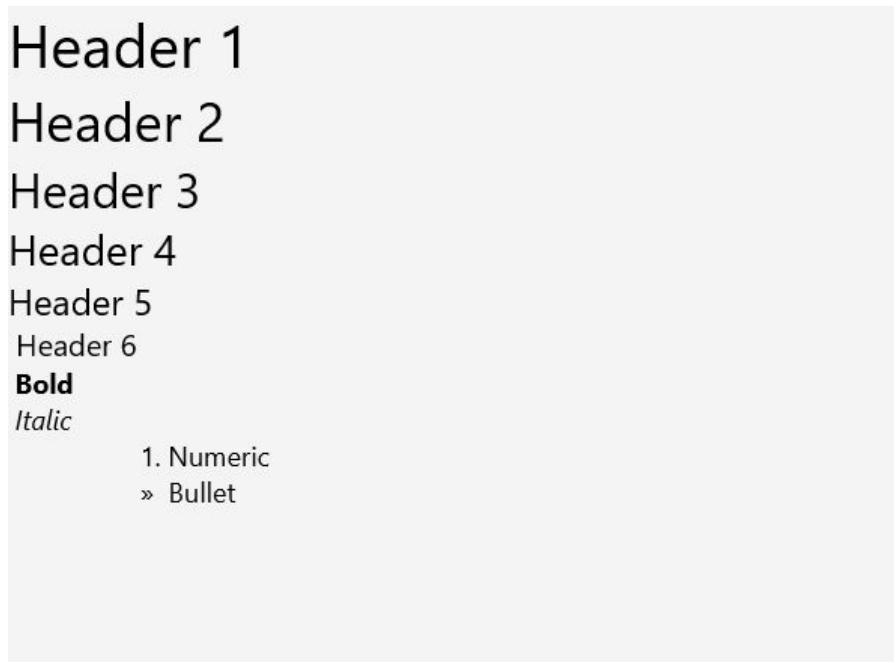
A continuación se pondrá el contenido del “Markdown Use Example” en la aplicación y se mostrará como quedará en la vista “Preview”.

Contenido de Example en la aplicación:



```
Markdown text editor
File Preview Help
#Header 1
##Header 2
###Header 3
####Header 4
#####Header 5
#####Header 6
* Bold *
/ Italic /
1. Numeric
+ Bullet
```

Preview del formato aplicado al contenido:



Header 1
Header 2
Header 3
Header 4
Header 5
Header 6
Bold
Italic
1. Numeric
» Bullet

Aquí un vídeo con una demostración básica de la aplicación:

<https://www.youtube.com/watch?v=AJqykC6Exbs>