Diseño de un Analizador Léxico para Markdown

Memoria de Práctica - Modelos de Computación

David Bacas Posadas
Julián Carrión Tovar
José Ángel Carretero Montes

24 de diciembre de 2025

Índice

1. Introducción

El objetivo de este proyecto es la implementación de un procesador de textos que transforme documentos en formato *Markdown* enriquecido con LaTeX a formato HTML5. Para ello, se ha utilizado la herramienta **flex**, basando el diseño del escáner en la teoría de Autómatas Finitos y Expresiones Regulares.

2. Limitaciones

Suponemos un uso de Markdown muy sencillo en el que únicamente podrán realizarse acciones básicas como inserción de títulos, escritura en cursiva, escritura en negrita, inserción de hipervínculos, inserción de listas enumeradas e inserción de bloques de ecuaciones. Así pues, este programa no será compatible con todo lo relacionado de Markdown que se salgan de las reglas escritas anteriormente, como por ejemplo: inserción de fotos, no por inviabilidad sino por simplificación del problema.

3. Utilización del programa

El proyecto incluye un archivo `Makefile` para facilitar el proceso de compilación y ejecución. A continuación se detallan las instrucciones para utilizar el programa.

3.1. Compilación

Para generar el ejecutable, abra una terminal en el directorio del proyecto y ejecute el comando:

```
make
```

Este comando realizará automáticamente los siguientes pasos:

1. Generará el código fuente C++ (`lex.yy.cc`) a partir del archivo Flex (`procesador.l`).
2. Compilará el código generado utilizando `g++` para crear el ejecutable llamado **procesador**.

3.2. Ejecución

El programa funciona mediante la lectura de la entrada estándar y la escritura en la salida estándar. Para procesar un archivo Markdown, utilice los operadores de redirección de la consola:

```
./procesador < prueba.md > archivo_salida.html
```

Donde `prueba.md` es el fichero fuente en Markdown/LaTeX y `archivo_salida.html` es el archivo HTML resultante.

3.3. Pruebas Automáticas

El archivo `Makefile` incluye un objetivo de prueba predefinido. Para ejecutar una prueba rápida con el archivo `prueba.md`, utilice:

```
make test
```

Esto generará automáticamente un archivo `salida.html` y mostrará un mensaje de confirmación.

3.4. Limpieza

Para eliminar los archivos generados (el ejecutable, el código fuente intermedio y los archivos de salida de prueba), ejecute:

```
make clean
```

4. Formalización Teórica

Para la descripción formal de los patrones, utilizaremos las siguientes operaciones sobre lenguajes regulares:

- **Unión ($+$):** $r + s$ denota el lenguaje $R \cup S$.
- **Concatenación (rs):** Denota el lenguaje RS.
- **Clausura ($*$):** r^* denota el lenguaje R^*.
- **Conjunto Σ:** Representa el alfabeto de caracteres ASCII imprimibles.

5. Análisis de Patrones y Autómatas

A continuación, se formalizan los patrones del analizador, presentando su expresión regular teórica y el Autómata Finito Determinista (AFD) correspondiente.

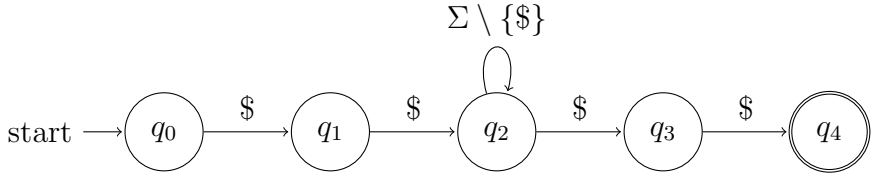
5.1. Ecuaciones en Bloque (Display Math)

Detecta bloques de ecuaciones centradas delimitadas por doble símbolo de dólar.

- **Definición en Flex:** `\$\$[^$]*\$\$`
- **Expresión Regular Formal:** Sea d el símbolo '$'.

$$r_{bloque} = dd \cdot (\Sigma \setminus \{d\})^* \cdot dd$$

Autómata Finito:



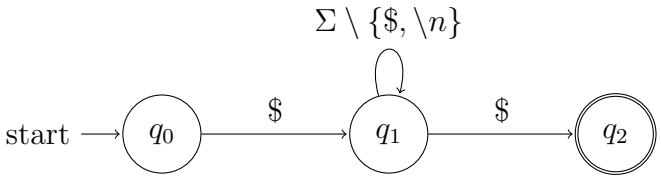
5.2. Ecuaciones en Línea (Inline Math)

Fórmulas insertadas en el texto delimitadas por un solo dólar.

- **Definición en Flex:** `\$[^$\n]*\$`
- **Expresión Regular Formal:** Sea d el símbolo '$' y n el salto de línea.

$$r_{inline} = d \cdot (\Sigma \setminus \{d, n\})^* \cdot d$$

Autómata Finito:



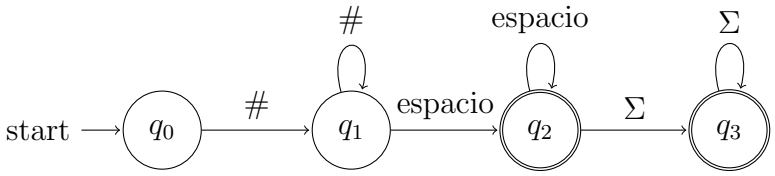
5.3. Encabezados (Títulos)

Secuencia de 1 a 6 almohadillas al inicio de línea. Suponemos que el formato de Markdown es correcto por lo que el límite de almohadillas es 6.

- **Definición en Flex:** `^#{1,6}[ ]+.*`
- **Expresión Regular Formal:** Sea $h = \#$ y $e =$ espacio.

$$r_{header} = (h + hh + \cdots + h^6) \cdot e \cdot e^* \cdot \Sigma^*$$

Autómata Finito (Simplificado):



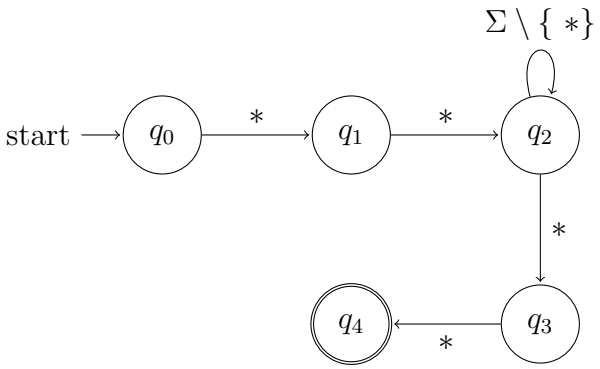
5.4. Negrita (Bold)

Texto entre doble asterisco.

- **Definición en Flex:** `\*\*[^*]*\*\*`
- **Expresión Regular Formal:** Sea $a = *$.

$$r_{bold} = aa \cdot (\Sigma \setminus \{a\})^* \cdot aa$$

Autómata Finito:



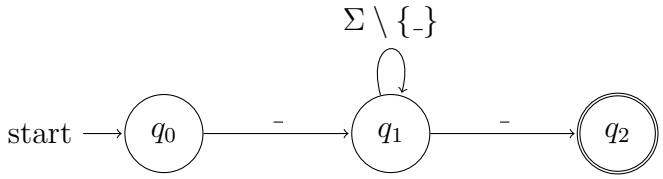
5.5. Cursiva (Italic)

Texto entre guiones bajos.

- **Definición en Flex:** `_[^_]*_`
- **Expresión Regular Formal:** Sea $u = \_$.

$$r_{italic} = u \cdot (\Sigma \setminus \{u\})^* \cdot u$$

Autómata Finito:



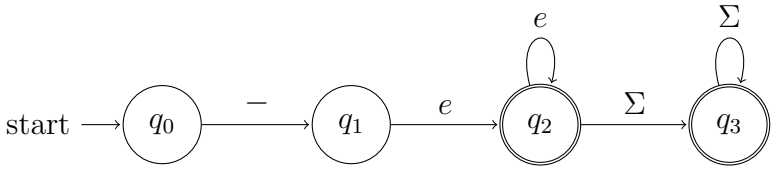
5.6. Elementos de Lista

Líneas que comienzan con un guión y un espacio.

- **Definición en Flex:** `^-[ ]+.*`
- **Expresión Regular Formal:** Sea $g = -$ y $e =$ espacio.

$$r_{list} = g \cdot e \cdot e^* \cdot \Sigma^*$$

Autómata Finito:



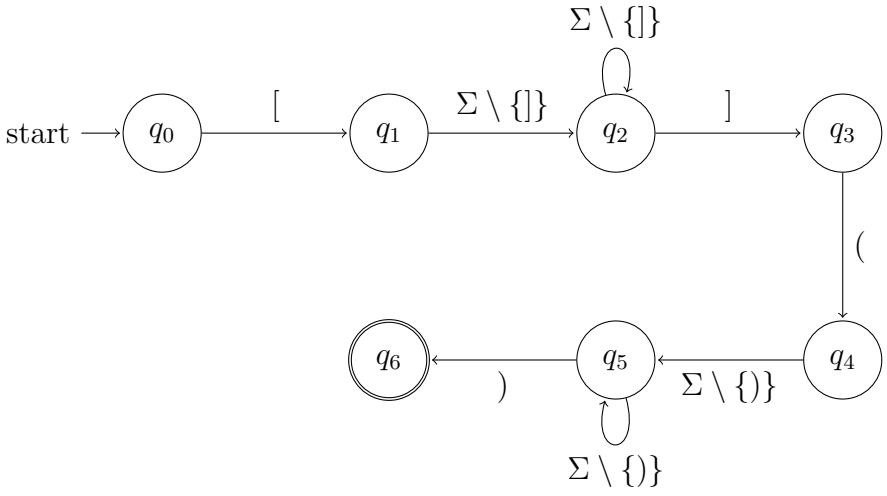
5.7. Enlaces (Links)

Reconoce el formato estándar de Markdown `[texto](url)`.

- **Definición en Flex:** `\[([^]]+)\]\(([^)]+)\)`
- **Expresión Regular Formal:**

$$r_{link} = [\cdot(\Sigma \setminus \{]\})^+\cdot] \cdot (\cdot(\Sigma \setminus \{)\})^+\cdot)$$

Autómata Finito:



6. Implementación Técnica

6.1. Resolución de Conflictos

En flex, el sistema resuelve ambigüedades basándose en dos reglas fundamentales:

1. **Longitud:** Se selecciona la regla que empareja la cadena más larga posible (Principio *Longest Match*).
2. **Prioridad:** Si las longitudes son iguales, se selecciona la regla que aparece primero en el fichero de definición.

6.2. Código Fuente (Extracto)

La implementación en C++ utiliza la clase `yyFlexLexer`. A continuación se muestra cómo las expresiones regulares se traducen a la sintaxis de flex en el fichero `.l`:

```
/* Definiciones (Alias) */
LATEX_BLOQUE    \$\$[^$]*\$\$
LATEX_INLINE    \$[^$\n]*\$
HEADER          ^#{1,6}[ ]+.*
BOLD            \*\*[^*]*\*\*
ITALIC          _[^_]*_
LIST_ITEM       ^-[ ]+.*
LINK            \[([^]]+)\]\(([^)]+)\)

%%
{LATEX_BLOQUE} {
    cout << "<div␣class='math'>" << yytext << "</div>";
}
{HEADER} {
    // Logica para contar # y generar <Hn>
}
```

7. Conclusión

La práctica demuestra la aplicación directa de la teoría de lenguajes formales. Hemos comprobado que, dado un lenguaje regular descrito por una expresión r, es posible construir un autómata finito equivalente que actúe como reconocedor. La herramienta flex automatiza la construcción del AFD subyacente, permitiendo un procesamiento eficiente del texto.