



UNIVERSIDAD SERGIO ARBOLEDA

FLEX & BIZON

DOCENTE

JOAQUIN SANCHEZ CIFUENTES

PRESENTADO POR

MARIO JIMENEZ LOPEZ

19 FEBRERO 2026

## EJEMPLO 1

En este primer ejemplo se trabajó con un programa básico escrito en Flex, el cual tiene como objetivo hacer una introducción sobre el funcionamiento de un analizador léxico simple. En este ejercicio se definieron reglas mediante expresiones regulares para reconocer patrones específicos en la entrada y ejecutar acciones asociadas a cada coincidencia.

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ nano fb1-1.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ flex fb1-1.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ ls
fb1-1.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ flex fb1-1.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ ls
```

Figura 1.1 – Ej1

El procedimiento realizado consistió en generar el código C a partir del archivo .l utilizando Flex, compilarlo con gcc y posteriormente ejecutar el programa desde la terminal. Durante la ejecución se ingresaron diferentes entradas de prueba para verificar el reconocimiento de los patrones definidos. Las capturas incluidas muestran el proceso de compilación, ejecución y los resultados obtenidos, evidenciando el correcto funcionamiento del analizador léxico.

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ gcc lex.yy.c -o fb1-1
-lfl
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ ls
fb1-1  fb1-1.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ ./fb1-1
Prueba numero uno de flex
de mario para la clase de joaquin
      2      12      61
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ S
```

Figura 1. – Ej1

## EJEMPLO 2

En este ejemplo se realizó una ampliación del escáner básico incluyendo el manejo de comentarios y la capacidad de ignorar espacios y tabulaciones. El objetivo fue mostrar cómo Flex puede filtrar entradas irrelevantes mientras procesa los tokens significativos. Las pruebas confirmaron que los comentarios eran correctamente ignorados y que los operadores y números se reconocían como tokens válidos, permitiendo un flujo más limpio de análisis léxico.

```

majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ nano fb1-2.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ flex fb1-2.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ ls
fb1-2.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ gcc lex.yy.c
-o fb1-2 -lfl
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ ls
fb1-2  fb1-2.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ ./fb1-2
colour
color
smart
elegant
hello
hola
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$

```

Figura 2.1 – Ej2

Se realizó una prueba ingresando variedad de datos para verificar la salida en cada uno de los casos

```

majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo2$ ./fb1-2
clever
smart
puerta
puerta

1
1

```

Figura 2.2 – Ej2

### EJEMPLO 3

El propósito de este ejemplo fue introducir el reconocimiento de patrones más complejos, utilizando expresiones regulares más avanzadas. Además, se mostró cómo asociar acciones para convertir cadenas de texto en valores numéricos. La ejecución de pruebas mostró que tanto números como símbolos son correctamente identificados y procesados, evidenciando la

flexibilidad de Flex para manejar diferentes tipos de tokens numéricos.

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ nano fb1-3.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ ls
fb1-3.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ flex fb1-3.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ ls
fb1-3.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ gcc lex.yy.c
-o fb1-3 -lfl
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ s
s: no se encontró la orden
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ ls
fb1-3  fb1-3.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ ./fb1-3
+
PLUS
NEWLINE
-
MINUS
```

Figura 3.1 – Ej3

Para la salida se evidencia como lo que ingresa flex lo identifica como token.

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo3$ ./fb1-3
/
DIVIDE
NEWLINE
*
TIMES
NEWLINE
1
NUMBER 1
NEWLINE
?
Mystery character ?
NEWLINE
m
Mystery character m
NEWLINE
█
```

Figura 3.2 – Ej3

#### EJEMPLO 4

Implementación de un escáner que no solo reconocía operadores y números, sino que también gestionaba caracteres desconocidos mediante una regla general. El objetivo fue demostrar cómo Flex maneja entradas inesperadas y cómo se pueden implementar acciones específicas para cualquier token no definido. Las pruebas realizadas mostraron que los números y operadores se reconocían correctamente, mientras que cualquier carácter no esperado activaba la regla de “Mystery character”, evidenciando el control total sobre la entrada del escáner

```
majilo@majiloPC: ~/Universidad/Lenguajes Programacion/Flex/Ejemplo4
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ nano fb1-4.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ flex fb1-4.l

majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ ls
fb1-4.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ gcc lex.yy.c
-o fb1-4 -lfl
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ ls
fb1-4  fb1-4.l  lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ ./fb1-4
12+5
258 = 12
259
258 = 5
264
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$
```

Figura 4.1 – Ej4

## EJEMPLO 5

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ls
fb1-5.y
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ nano fb1-5.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ bison -d fb1-5.y
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ls
fb1-5.l  fb1-5.tab.c  fb1-5.tab.h  fb1-5.y
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ flex fb1-5.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ gcc fb1-5.tab.c lex.yy.c -o fb1-5 -lfl
```

Figura 5.1 – Ej5

## Pruebas

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ nano fb1-5.l
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
1-5
= -4
3+1
= 4
m
error: syntax error
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$
```

Figura 5.2 – Ej5

## EJERCICIOS

### EJERCICIO 1

Se evaluó si la calculadora aceptaba líneas que contuvieran únicamente comentarios. El objetivo fue demostrar cómo el escáner y el parser manejan entradas que no generan tokens significativos. Las pruebas mostraron que, en su estado actual, el parser no aceptaba una línea con solo comentarios. Esto evidencia que es más sencillo resolverlo en el escáner, agregando reglas para ignorar comentarios antes de que lleguen al parser.

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ ls
Ejemplo1 Ejemplo2 Ejemplo3 Ejemplo4 Ejemplo5
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex$ cd Ejemplo5
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ls
fb1-5 fb1-5.l fb1-5.tab.c fb1-5.tab.h fb1-5.y lex.yy.c
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
bash: ./fb1-5: No existe el archivo o el directorio
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
5-2
= 3
//
error: syntax error
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
/
error: syntax error
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
apaasdads
error: syntax error
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$
```

Figura 1.2.1 – Ej1

El error ocurre porque flex al leer la entrada, no reconoce número o ningún operador válido y el parser se queda esperando “exp EOL” pero al no recibir la información válida entonces bison genera el error de sintaxis.

La calculadora no acepta una línea que contenga solo un comentario porque el parser requiere una expresión válida. El comentario no genera un token válido, por lo que el parser produce un error de sintaxis.

### EJERCICIO 2

Se modificó el escáner para reconocer números hexadecimales usando el patrón `0x[a-fA-F0-9]+` y convertirlos a enteros con `strtoul`. El parser se ajustó para aceptar estos tokens como `NUMBER`. Además, la salida se modificó para mostrar el resultado tanto en decimal como en hexadecimal. El objetivo fue ampliar la calculadora para soportar distintos sistemas numéricos, y las pruebas confirmaron el correcto reconocimiento y cálculo de números decimales y hexadecimales.

Se cambia la línea de

```
calclist:
    | calclist exp EOL { printf("= %d\n", $2); }
    ;
```

Figura 2.2.1 – Ej2

Por

```
calclist:
    | calclist exp EOL { printf("= %d (0x%X)\n", $2, $2); }
    ;
```

Figura 2.2.2 – Ej2

Se modificó la línea de salida para mostrar el resultado en decimal y hexadecimal (0x%X), con el fin de facilitar la verificación de los operadores binarios (AND, OR) que se implementarán en el ejercicio extra, ya que los resultados de estas operaciones son más claros y legibles en formato hexadecimal.

```
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
10+5
= 15 (0xF)
0xA+5
= 15 (0xF)
0xA+0x5
= 15 (0xF)
prueb
error: syntax error
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$
```

Figura 2.2.3 – Ej2

### EJERCICIO 3

El símbolo | ya estaba definido en la gramática como un operador unario para calcular el valor absoluto. Si se intentara reutilizar el mismo símbolo como operador binario OR (por ejemplo, en una regla como `exp ABS factor`, el parser no podría distinguir cuándo | debe interpretarse como operador unario y cuándo como operador binario. Esta doble interpretación genera una ambigüedad en la gramática, ya que una misma secuencia de entrada podría analizarse de más de una manera. Por esta razón, se decidió utilizar un símbolo diferente (||) para el operador OR, evitando así la ambigüedad y manteniendo la gramática clara y no conflictiva.

```

majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$ ./fb1-5
| -5
= 5 (0x5)
5||3
= 7 (0x7)
5&3
= 1 (0x1)
0xA+5
= 15 (0xF)
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo5$

```

Figura 3.2.1 – Ej3

## EJERCICIO 4

Para comprobar el comportamiento del escáner generado con Flex en fb1-4.1, se realizaron pruebas con distintas entradas. Al ingresar una suma basica el programa reconoce correctamente los tokens NUMBER, ADD y EOL. En cambio, al ingresar 12a, reconoció el número 12 y luego activó la regla general. mostrando el mensaje “Mystery character a”. Esto demuestra que Flex maneja explícitamente los caracteres no definidos mediante reglas específicas. Una versión escrita manualmente podría comportarse de forma diferente dependiendo de su implementación, por lo que no se puede asegurar que ambos escáneres reconozcan exactamente los mismos tokens en todos los casos.

```

majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ ./fb1-4
12+3
258 = 12
259
258 = 3
264
12a
258 = 12
Mystery character a
264
majilo@majiloPC:~/Universidad/Lenguajes Programacion/Flex/Ejemplo4$ █

```

Figura 4.2.1 – Ej4

## EJERCICIO 5

Flex no es una herramienta adecuada cuando el análisis léxico requiere un fuerte manejo de contexto o estructuras que no pueden describirse fácilmente mediante expresiones regulares. Por ejemplo, lenguajes donde la indentación afecta la sintaxis o donde existen dependencias complejas entre tokens pueden hacer que el escáner generado con Flex sea difícil de implementar y mantener. Esto se debe a que Flex está diseñado principalmente para reconocer lenguajes regulares.

## **EJERCICIO 6**

Flex no es una herramienta adecuada cuando el análisis léxico requiere un fuerte manejo de contexto o estructuras que no pueden describirse fácilmente mediante expresiones regulares. Por ejemplo, lenguajes donde la indentación afecta la sintaxis o donde existen dependencias complejas entre tokens pueden hacer que el escáner generado con Flex sea difícil de implementar y mantener. Esto se debe a que Flex está diseñado principalmente para reconocer lenguajes regulares.