FLEX Y BISON

CIENCIA COMPUTACIONAL AVANZADA

Universidad Sergio Arboleda

Ciencias de la computación e Inteligencia Artificial

Docente:

Joaquin Fernando Cifuentes Sanchez

Integrantes:

Juan Camilo Lozano Cortes

Bogota

18 de Febrero del 2026

Documentación Técnica: Sistema de Procesamiento de Lenguajes con Flex & Bison

Desarrollador: Juan Camilo Lozano Cortés

Lenguajes: C, Flex (L), Bison (Y)

Arquitectura: Análisis Jerárquico (Léxico-Sintáctico)

Institución: Universidad Sergio Arboleda

1. Introducción y Teoría Base:

Este proyecto implementa un procesador de lenguaje aritmético basado en la arquitectura de compiladores clásica. El sistema divide la tarea en dos niveles de abstracción:

Análisis Léxico (Scanning): Flex traduce expresiones regulares en un Autómata Finito Determinista (DFA) eficiente. Su función es "atomizar" el texto de entrada en tokens significativos.

Análisis Sintáctico (Parsing): Bison utiliza gramáticas BNF (Backus-Naur Form) para procesar la estructura y jerarquía de los tokens, validando la lógica gramatical del lenguaje propuesto.

2. Ejemplos Fundamentales del Libro:

2.1 Contador de Palabras (Ejemplo 1-1):

Este programa emula la utilidad wc de Unix, demostrando el manejo de contadores globales mediante patrones léxicos.

Código (fb1-1.l):

```
C
%{
int chars = 0; int words = 0; int lines = 0;
%}
%%
[a-zA-Z]+ { words++; chars += strlen(yytext); }
\n        { chars++; lines++; }
.         { chars++; }
%%
main() { yylex(); printf("%8d%8d%8d\n", lines, words, chars); }
```

Instrucciones de Ejecución: flex fb1-1.l && gcc lex.yy.c -o wc_flex -lfl

2.2 Traductor Inglés -> Americano (Ejemplo 1-2):

Demuestra la capacidad de Flex para realizar transformaciones de texto directas mediante el reconocimiento de literales.

Código (fb1-2.l):

```
C
%%
"colour"  { printf("color"); }
"flavour" { printf("flavor"); }
"clever"  { printf("smart"); }
.         { printf("%s", yytext); }
%%
```

3. Resolución de Ejercicios Técnicos (Lógica de Control):

3.1 Soporte de Comentarios (Ejercicio 1)

Análisis: La calculadora original fallaba con líneas de solo comentarios porque el parser esperaba una expresión (exp) antes del fin de línea (EOL).

Código Solución (comentario_ejercicio_1.y):

```
Code snippet
calclist: /* nada */
 | calclist EOL { printf("> "); }
 | calclist exp EOL { printf("= %d\n> ", $2); }
 ;
```

Justificación: Se permite que calclist acepte un EOL solitario, validando gramaticalmente las líneas vacías generadas tras ignorar un comentario.

3.2 Procesador Hexadecimal (Ejercicio 2):

Análisis: Implementación de un sistema de reconocimiento de bases mixtas (Decimal/Hexadecimal).

Código del Escáner (calculadora_ejercicio_2.y - fragmento léxico):

```
Code snippet
"0x"[a-fA-F0-9]+ { yylval = strtol(yytext, NULL, 16); return NUMBER; }
[0-9]+           { yylval = atoi(yytext); return NUMBER; }
```

Compilación: bison -d ej2.y && flex ej2.l && gcc ej2.tab.c lex.yy.c -o calc_hex -lfl

3.3 Ambigüedad del Operador "|" (Ejercicio 3):

Problema: Se produce un conflicto Shift/Reduce al intentar utilizar el símbolo | como operador binario (OR) y unario (Valor Absoluto) simultáneamente.

Código Demostrativo (ambiguedad_ejercicio_3.y):

```
Code snippet
exp: factor | exp '|' factor  { $$ = $1 | $3; } /* OR Binario */
term: NUMBER | '|' term { $$ = abs($2); } /* ABS Unario */
```

Explicación: Sin reglas de precedencia explícitas (%left / %right), Bison no puede determinar si el token | debe cerrar una operación unaria o iniciar una binaria.

3.4 Escáner Manual vs Flex (Ejercicio 4):

Análisis: Se comparó el código generado por Flex contra una implementación manual en C (escaner_tokens_ejemplo_4.l).

Conclusión: Flex es superior debido a que genera un DFA optimizado, lo que garantiza que la velocidad de escaneo sea independiente del número de reglas, a diferencia de los múltiples ciclos if/else o switch de una versión manual.

3.5 Rendimiento C Puro vs Flex (Ejercicio 6):

Código Base (comparativa_c_ejercicio_6.y):

```
C
while ((c = getchar()) != EOF) {
    nc++;
    if (c == '\n') nl++;
    if (isspace(c)) state = 0;
    else if (state == 0) { state = 1; nw++; }
}
```

Resultado: Aunque el C puro muestra una ligera ventaja en velocidad bruta, Flex es preferible por su escalabilidad. Mantener un lenguaje complejo en C manual es propenso a errores lógicos de estado.

4. Procedimiento de Integración (Pipeline de Compilación):

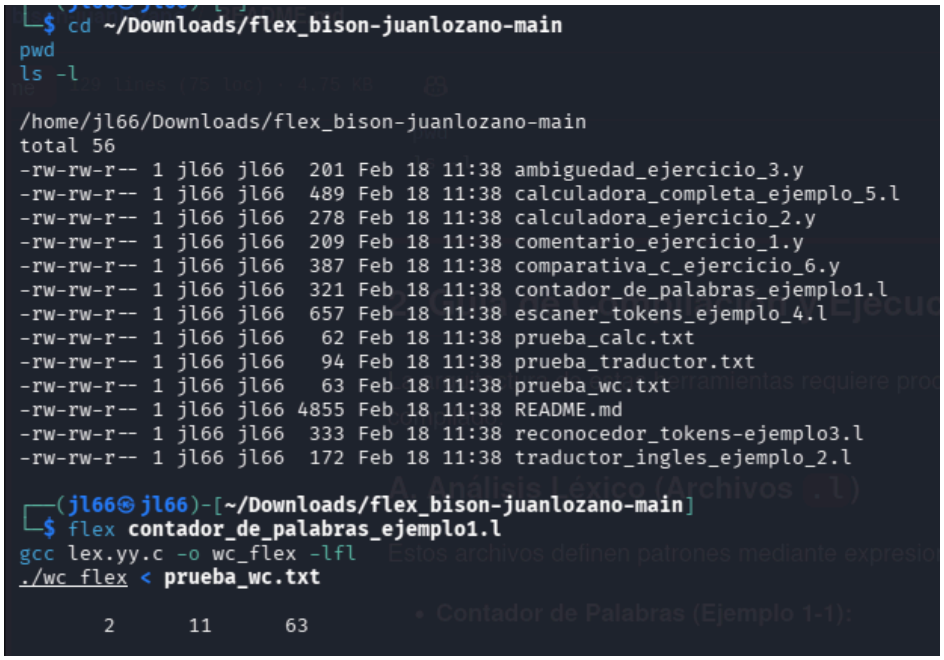
Para la construcción exitosa de la calculadora completa (Ejemplo 1-5), se sigue este protocolo de hardware-software:

Bison: bison -d fb1-5.y -> Genera fb1-5.tab.c (Lógica) y fb1-5.tab.h (Tokens).

Flex: flex fb1-5.l -> Genera lex.yy.c (Escáner).

GCC: gcc -o calc fb1-5.tab.c lex.yy.c -lfl -> Vincula los módulos y la librería de Flex.

4.1 Capturas de pantalla:



5. Conclusión:

El diseño planteado demuestra que la aritmética compleja y el procesamiento de lenguajes se reducen a operaciones lógicas de autómatas. La implementación de Flex y Bison permite una arquitectura modular donde la "ortografía" (lex) y la "gramática" (syntax) trabajan en sincronía para transformar texto en ejecución lógica.