# UNIVERSIDAD DE GUADALAJARA

**CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍA**
**DIVISION DE ELECTRONICA Y COMPUTACION**

**DEPARTAMENTO DE CIENCIAS COMPUTACIONALES**


# SEMINARIO DE SOLUCIÓN DE PROBLEMAS DE TRADUCTORES DE LENGUAJE I (I7026)

## Reporte de Actividad Práctica

(reporte de investigación, síntesis de lectura o cuestionario)

## Práctica 5: "Análisis Léxico"


**Alumno: Torres Rivera Rodrigo**
**Código: 397423431**

Sección: D01

Profesor: Valentín Martínez López


Fecha 11 de Noviembre de 2025
firma de revisado

## Nombre de la Práctica: Analizador lexico

## Objetivo de la Práctica:

Desarrollar un programa en lenguaje ensamblador que permita al usuario ingresar una cadena de texto desde el teclado, la cual representará una ecuación aritmética con datos de 16 bits con signo. La ecuación debe incluir al menos cuatro operandos y debe soportar las cuatro operaciones básicas: suma (+), resta (-), multiplicación (*) y división (/). El programa debe resolver la ecuación respetando la jerarquía de operaciones y mostrar el resultado en pantalla, también en formato de 16 bits con signo.

## Antecedentes:

Se requiere contar con el siguiente material:

-Computadora con sistema operativo compatible.

-Emulador EMU8086 instalado.

-Incluir la libreria "emu8086.inc", para las macro funciones

## Desarrollo:

### Características y Requisitos:

**1. Entrada de datos:**

- El programa debe leer una única vez la cadena de texto ingresada por el usuario desde el teclado.
- La cadena debe almacenarse en un arreglo para su posterior procesamiento.
- La longitud de la cadena debe permitir datos de un solo dígito.

**2. Operaciones aritméticas:**

- El programa debe soportar las cuatro operaciones básicas: suma, resta, multiplicación y división.
- Se debe respetar la jerarquía de operaciones (precedencia de operadores).

**3. Salida en pantalla:**

- El programa debe mostrar en pantalla el nombre del alumno.
- Debe imprimir instrucciones claras para el usuario sobre cómo introducir la cadena de la ecuación, incluyendo detalles como la base numérica y la cantidad de dígitos permitidos.
- El resultado de la ecuación debe mostrarse en formato de 16 bits con signo.

**4. Manejo de interrupciones:**

- Se permite el uso de interrupciones, excepto las interrupciones 21h y 80h.
- Para finalizar el programa, se debe utilizar la interrupción 20h.

**5. Registro SP (Stack Pointer):**

- El registro SP debe tener el valor `FFF8h` al finalizar el programa correctamente. Este valor indica que los procedimientos y el retorno al sistema operativo se han realizado de manera adecuada.
- Si el registro SP tiene este valor debido al uso incorrecto de instrucciones que lo modifiquen, pero no se cierran correctamente los procedimientos o el retorno al sistema operativo, la práctica será rechazada.

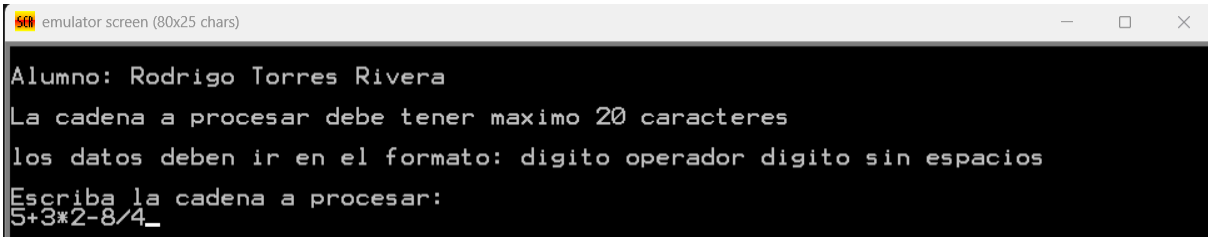
**6. Consideraciones adicionales:**

- El programa debe ser eficiente y manejar correctamente los errores, como divisiones por cero o entradas inválidas.
- Se recomienda consultar la documentación de ayuda del emu8086 para obtener información detallada sobre las interrupciones y su implementación.

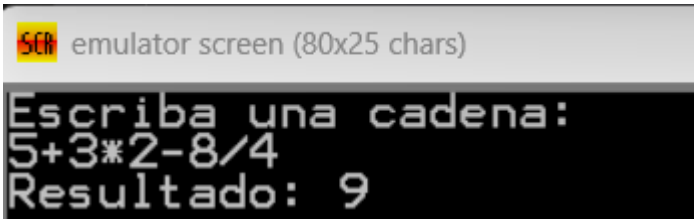
## Resultados Obtenidos:

- **Entrada:**
  - 1. El programa muestra en pantalla:
  - "Ingrese una ecuación aritmética con datos de 16 bits con signo (máximo 60 caracteres):"
  - 2. El usuario ingresa:
  - `"5+3*2-8/4"`
- **Salida:**
  - 3. El programa resuelve la ecuación respetando la jerarquía de operaciones y muestra el resultado:
  - "El resultado es: 9".
  - 4. Finalmente, el programa finaliza correctamente utilizando la interrupción 20h y asegurando que el registro SP tenga el valor `FFF8h`.

Introducimos el dato de prueba:




El resultado es 9 como era esperado:

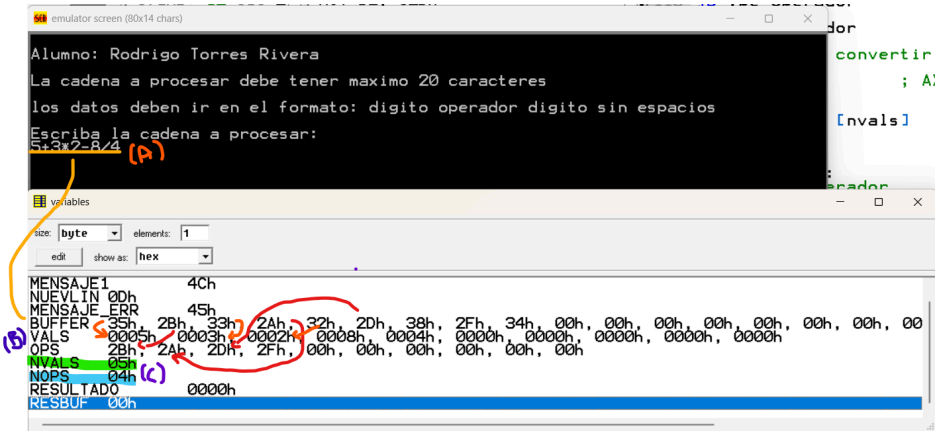



## .Conclusiones:

La sección de código etiquetada como programa principal resume el proceso para resolver la ecuación, una vez con la ecuación almacenada como una cadena de caracteres debemos verificar si cumple con el formato requerido para su análisis, en caso de ser errónea la volvemos a pedir (esto manejado dentro de la subrutina para evaluar la expresión línea 86 dentro de la subrutina), si la cadena es válida, procedemos a parsear la expresión, esto genera variables auxiliares *Vals (almacenar los valores numéricos) Ops(los símbolos de los operadores) Nvals (cantidad de valores numéricos) Nops(total de operadores a procesar)* para en la siguiente etapa evaluar la expresión es donde aplicamos la resolución de la ecuación acorde al orden de precedencia de los operadores esto implica que se deba realizar varias “pasadas” por la cadena de la ecuación para que cada vez vayamos realizando las operaciones y acumulemos el resultado .

```
448     ;leer los datos en buffer
449     lea di, buffer
450     mov dx, tammax
451     call get_string
452
453     ;Validar la expresión ingresada: solo dígitos y operadores (+ - * /)
454     ;Formato requerido: digito operador digito operador ... (termina en dígito)
455     lea si, buffer
456     call validar_expresion
457
458     ; Parsear la expresión a arrays vals[] y ops[]
459     lea si, buffer
460     call parsear_expresion
461
462     ; Evaluar con precedencia de operadores
463     call evaluar_expresion
464
465     ; Imprimir resultado
466     call imprimir_resultado
467
468     ;printn ""
469     ;printn "La cadena escrita fue:"
470     ;lea si,buffer
471     ;call print_string
472     ;printn ""
473
474     ;Salir al Sistema Operativo
475     mov ax, 4c00h
476     int 21h
477     ret
```

Primero debemos “parsear” la cadena de caracteres en la siguiente imagen:

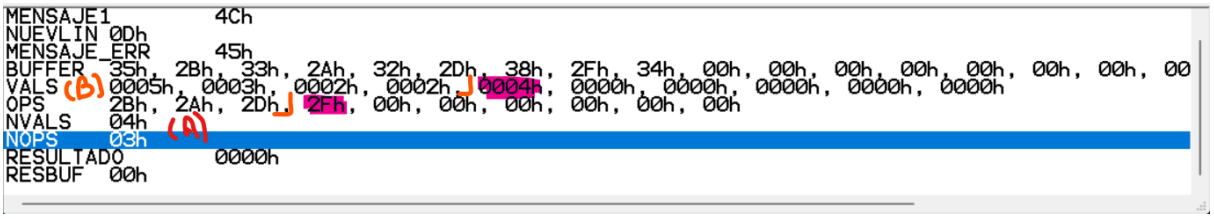



la *variable Buffer* contiene la cadena (A), las *variables Vals y Ops* contienen los datos ya convertidos en binario y los operadores respectivamente, las *variables Nvals y Nops* contienen el número de elementos de cada arreglo respectivamente, esto ayuda a calcular la longitud del arreglo multiplicando el tamaño de cada dato (DW en este caso) y al sumarlo a la dirección de inicio del arreglo [Vals] o [Ops] podemos obtener la dirección del final del arreglo. El siguiente paso es “evaluar la expresión” donde realizaremos las operaciones siguiendo el orden de presencia división primero, seguida de multiplicación, resta y por último la suma.

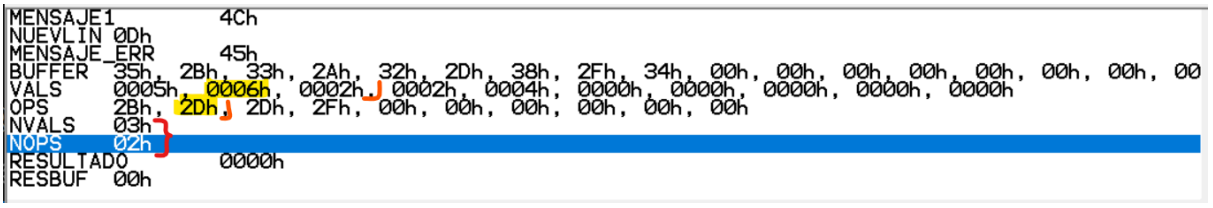
En la etapa de evaluar la expresión veremos como se van “compactando” los arreglos con los resultados de las operaciones intermedias, por ejemplo comenzamos con los arreglos sin alteración como se muestra en la figura siguiente:

```
MENSAJE1        4Ch
NUEVLIN 0Dh
MENSAJE_ERR     45h
BUFFER  35h, 2Bh, 33h, 2Ah, 32h, 2Dh, 38h, 2Fh, 34h, 00h, 00h, 00h, 00h, 00h, 00h, 00h, 00
VALS    0005h, 0003h, 0002h, 0008h, 0004h, 0000h, 0000h, 0000h, 0000h, 0000h
OPS     2Bh, 2Ah, 2Dh, 2Fh, 00h, 00h, 00h, 00h, 00h, 00h
NVALS   05h
NOPS    04h
RESULTADO       0000h
RESBUF  00h
```

Tras realizar el primer “pase” para efectuar todas las divisiones, tenemos que las cadenas cambiaron a:

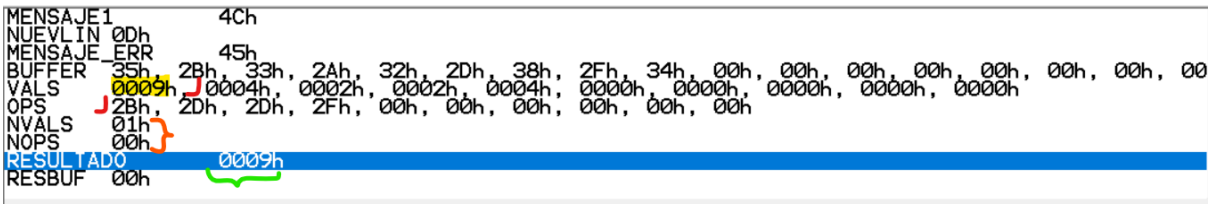



Donde (A) muestra que el número de elementos de cada arreglo (denotado por Nvals y Nops), se redujo en una unidad para Nops, donde el valor de la división es escrito en el arreglo Vals en el penúltimo elemento y Nops es “truncado” al penúltimo elemento también. Tras evaluar la operación multiplicación:



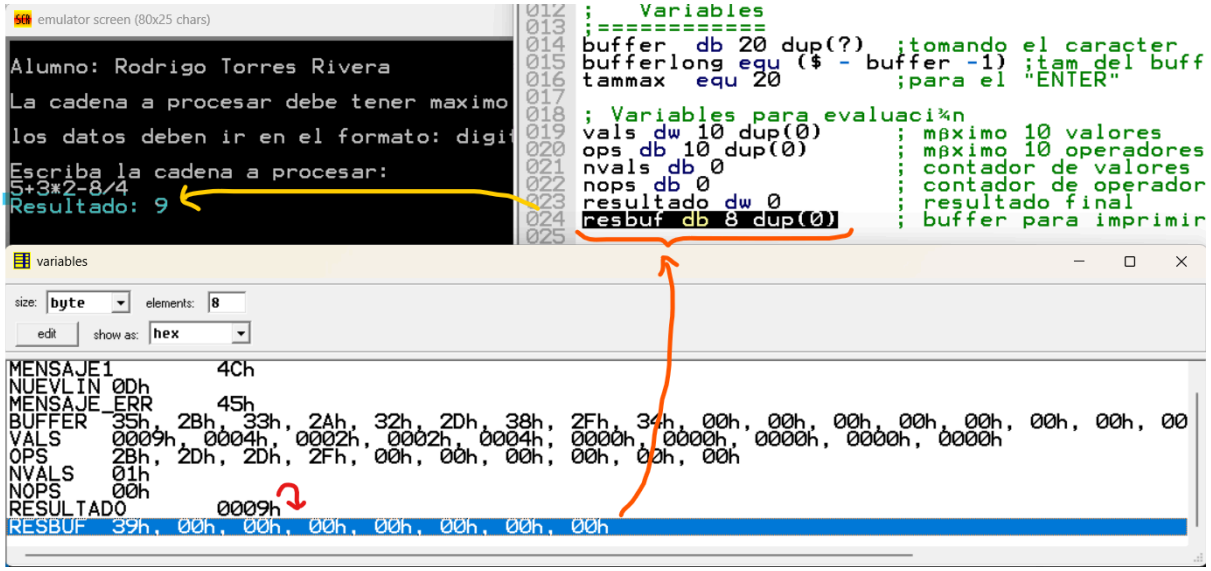

Ahora vemos que el símbolo de la operación multiplicación fue removido del arreglo *Ops* y que en *Vals* se sustituyó el valor de la operación la casilla correspondiente y se recorrieron los datos restantes, por lo que las variables *Nvals y Nops* redujeron su número de elementos.


Tras procesar la operación de resta, observamos que el único símbolo que falta es el de la suma y que nuestros datos se han reducido a sólo dos donde uno corresponde al resultado acumulado hasta ahora. El programa procede a ejecutar la última operación:




El arreglo *Ops* se ha quedado vacío indicando que ya no quedan operaciónes que realizar como indica *Nops* así *Vals* contendrá solo un elemento, el cual es el resultado acumulado de las operaciones, por lo que se copia a la variable *Resultado*.

Por último este valor es convertido a su representación en ASCII en la variable *Resbuf*:




Como nota adicional observe que *Resbuf* se definió tomando en cuenta que su representación podría requerir 8 cifras significativas (siete si  el resultado es negativo).