# Teoría de la computación - Práctica 3

**Grupo Lunes 8:00-10:00 semanas B**
**— Práctica 3 —**
**Autor:** Fabrizio Duarte
**NIP:736857**
**Autor:** Fernando Pastor Peralta
**NIP:897113**

**ATENCIÓN**: Estos programas no han sido probados en lab000, debido a la imposibilidad de uso del sitio.

**Ejercicio 1:**

El objetivo de este ejercicio era el análisis de los ficheros calcOrig.l y calcOrig.y mediante la compilación y ejecución de los mismos

**GRAMÁTICA calcOrig.y:**
**- Regla Inicial (calclist)**:
Representa una lista de cálculos.
Calclist→ϵ | calclist exp EOL
-Esta regla permite manejar múltiples expresiones separadas por líneas (`EOL`).
La lista de cálculos puede estar vacía (ϵ) o contener una o más expresiones.

**- Expresiones (`exp`)**:

Representa una expresión.
exp→factor | exp ADD factor | exp SUB factor
-Una expresión puede ser:
Un `factor` o una suma (`ADD`) o una resta (`SUB`) entre una expresión y un factor.

**- Factores (`factor`)**:

Representa un factor en una operación.
factor→term | factor MUL term | factor DIV term
-Un factor puede ser:
Un `term` o una multiplicación (`MUL`) o división (`DIV`) entre un factor y un término.

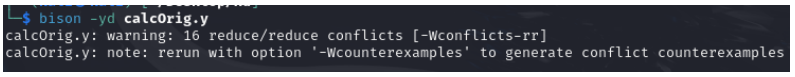
**- Términos (`term`)**:

Representa un término (el elemento más básico de una expresión).

term→NUMBER | OP exp CP
-Un término puede ser :
Un número (`NUMBER`) o una expresión entre paréntesis (`OP` ... `CP`).

**TOKENS:**
`ADD`: Representa el operador `+`.
`SUB`: Representa el operador `-`.
`MUL`: Representa el operador `*`.
`DIV`: Representa el operador `/`.
`NUMBER`: Representa un número.
`OP` y `CP`: Representan los paréntesis `(` y `)`.
`EOL`: Representa el fin de una línea.

**MENSAJES BISON:**




Al compilar el programa calcOrig.y usando bison se ha detectado 16 avisos que nos informa de conflictos reducción/reducción.
Esto ocurre por la ambigüedad en la gramática, en este caso dos o más reglas pueden aplicarse a la misma secuencia de entrada, lo que ocasiona el conflicto.
Para operaciones triviales el programa funciona correctamente, sin embargo para otras como sumas o restas seguidas da error. (Por ejemplo, 80/4/2 debería dar 10 pero da como resultado 40 ya que puede otorgar preferencia a la parte derecha de la operación).

La versión mejorada (`calcMejor`) corrige los problemas de precedencia, asociatividad y manejo de paréntesis que causaban errores en la versión original (`calcOrig`). Esto se logra mediante una gramática más clara, modular y jerárquica.

**Ejercicio 2.1:**

**1.Resumen:**

Modificamos la calculadora para que acepte cualquier número entero en base decimal o en otra base b entre 2 y 10, siguiendo los requerimientos del enunciado.

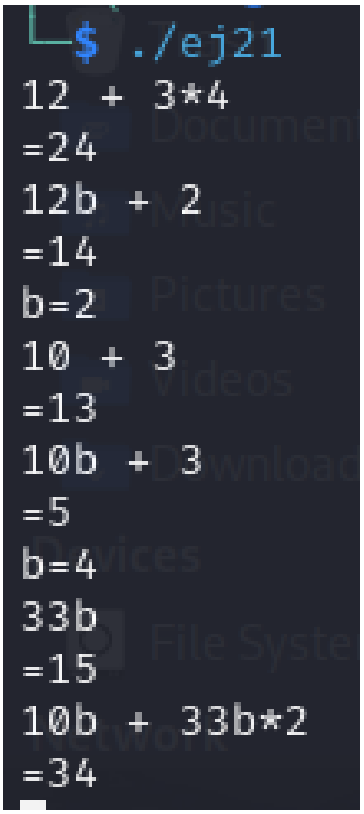
En primer lugar, creamos una variable b (base 10 por defecto), cuyo valor podremos cambiar durante la ejecución del programa si escribimos b = y un número entre 2 y 10.

Para registrar la instrucción anterior, hemos añadido dos token: B (base, usado al encontrar “b”) y EQ (equal, usado al encontrar “=”), además, hemos añadido a la regla “calclist” la posibilidad de encontrar una línea con el formato mencionado, esta nueva condición asigna el valor adecuado a nuestra base :

“| calclist B EQ NUMBER EOL {b = $4;}”

Para cambiar internamente el formato del número en base b, lo hacemos directamente en el código de flex. Al detectar un número que acabe en b ([0-9]+"b"), un código en c transforma dicho número de la base en la que está a decimal. Pasamos el token NUMBER con yylval como el número transformado en base decimal.

**2.Pruebas:**



**Ejercicio 2.2:**

**1.Resumen:**

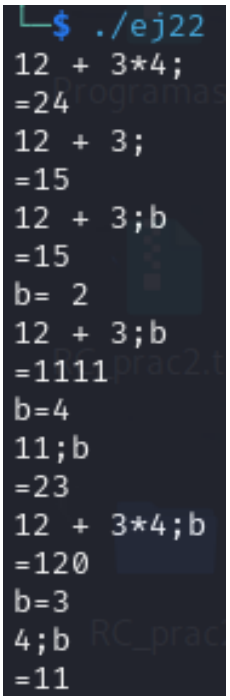
Modificamos la calculadora para que las líneas de entrada que acaben en “;” tengan un resultado decimal y que las líneas que acaban con “;b;” tenga el resultado en la base b.

Al igual que en el ejercicio anterior, creamos la variable b, con mismas características y los tokens  B y EQ (EQU).

Añadimos los tokens BASE10, para las operaciones con “;” (flex manda el token cuando encuentra ;) y “BASEB”, para las operaciones con “;b” (flex manda el token cuando encuentra ;b).

Para la impresión correcta en ambos formatos, modificamos calclist:
- Puede no haber nada.
- Puede que nos encontremos con una línea que acabe en ; (token BASE10), en cuyo caso tendremos que realizar la operación con normalidad, sin cambios:
    “| calclist exp BASE10 EOL { printf("=%d\n", $2); }”
-Puede que nos encontremos con una línea que acabe en ;b (token BASEB), en cuyo caso tendremos que pasar el valor calculado a la base b dada, esto lo hacemos mediante código en c, después imprimimos el resultado según la base.
    “| calclist exp BASEB EOL {/*código en c*/“}”
-Puede que queramos modificar la base b, con b = y un número entre 2 y 10, como en el ejercicio anterior.

**2.Pruebas:**



**Ejercicio 2.3:**

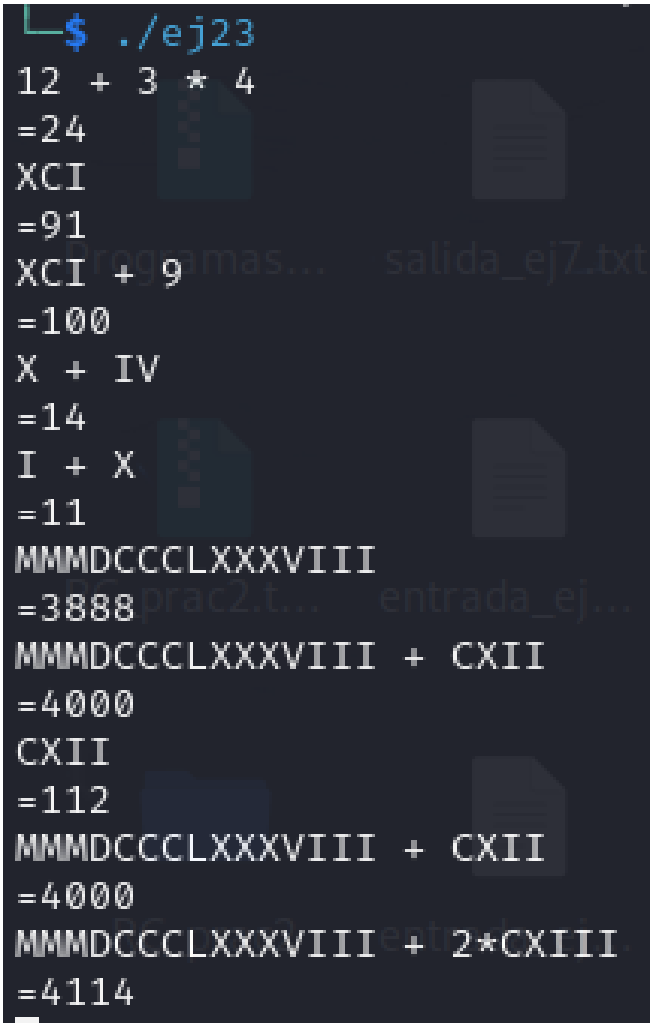
**1.Resumen:**

Modificamos la calculadora para que acepte enteros en decimal o en romano.

Añadimos el token ROMANO que será enviado por flex cada vez que se encuentre un número romano, en bison será procesado en la regla factorsimple.

Flex reconoce cadenas formadas únicamente por los caracteres válidos de los números romanos (`I, V, X, L, C, D, M`), con la regla `[IVXLCDM]+`, en el momento que se cumple esta regla, mediante código en c, traducimos el número en romano a número decimal, este número en decimal se le asignará a yylval en el momento de mandar el token ROMANO.

**2.Pruebas:**



**Ejercicio 3:**

**1.Resumen:**

Implementamos un analizador sintáctico para la gramática dada, en caso de que una cadena no pertenezca a ella, imprimimos el mensaje “syntax error”.

$$
\begin{aligned}
S &\rightarrow CxS \mid \epsilon \\
B &\rightarrow xCyy \mid xC \\
C &\rightarrow xBx \mid y
\end{aligned}
$$

Flex identifica los caracteres "x", "y", cuando esto sucede enviá los tokens x e y respectivamente a Bison. En flex, también ignoraremos los espacios y tabuladores, y en caso de que encontremos un fin de línea, se mandará el token EOL.

Con respecto al fichero de Bison, se han declarado los tokens del archivo de Flex y posteriormente las normas calclist (inicio), S, B y C.

El programa comienza verificando una regla inicial (calclist), que sirve como punto de partida para procesar una cadena, asegurándose de que inicia con el símbolo S y puede identificar el final de la cadena.

Después, el programa se desarrolla de forma recursiva, tal y como marca la gramática, permitiendo así identificar las distintas cadenas que conforman la misma.

**2.Pruebas**