

**Escuela Politécnica Nacional**



**Facultad de Ingeniería en Sistemas**

**Compiladores y lenguajes**

**GR1COM**

**Informe Proyecto Primer Bimestre**

**Grupo: #5**

**Integrantes**

* Pablo Xavier Sarzosa Virgilí
* Kevin Alexander Pérez Cruz
* Ricardo Josué Barros Tobar

**Profesor:** Ing. Rodrigo Chancusig MSc.

**Fecha de entrega:** 12/03/2021

**Resumen:**

En este documento se presentará las definiciones de expresión regular, los símbolos que componen para poder crear esta, la implementación en lex, la configuración y estructura que debe tener un archivo lex y yacc y por último una implementación de un programa combinando los sistemas. Se describirá cada parte de un compilador, así como también se procederá a analizar sus funciones, los símbolos que recibe y procesa. Con la ayuda del compilador de Ubuntu Linux se creará un programa que recibe una operación arimética. Para esto la línea ingresada deberá ser verificada mediante lex como símbolos para así convertirlos en una expresión arimética que la computadora entienda (yacc). El programa recibirá cinco tipos de sistemas de numeración, estos son: Binarios, decimales, hexadecimales, octales y romanos. También se analizarán símbolos de agrupación y símbolos aritméticos como suma, resta, multiplicación, división, potenciación, raíz cuadrada, operaciones trigonométricas, logarítmicas y exponenciales. Finalmente, el programa mostrará la respuesta de la expresión arimética ingresada en los sistemas de numeración ya mencionados.

**Introducción:**

Este informe es la parte terminada del proyecto del curso Compiladores y Lenguajes. Su primera fase fue presentada en el primer bimestre del semestre 2020-B. Para realizar el proyecto se hizo uso del program lex y yacc con ayuda del compilador de C incorporado en los sistemas operativos como Ubuntu Linux y sus derivados. El programa LEX es comúnmente utilizado por el programa YACC que es utilizado para reconocer expresiones regulares. El proyecto en su primera fase se basa en el reconocimiento de expresiones aritméticas como: suma, resta, multiplicación, división, potenciación raíz cuadrada también el programa puede reconocer entre símbolos de agrupación como son: paréntesis, llaves, corchetes, comillas francesas. Además, el programa permite el reconocimiento de distintos sistemas de numeración, entre ellos están el sistema de numeración binario, decimal, hexadecimal, octal y romano. Para agregar, el programa reconocerá número en una longitud máxima de -10000 a 10000. En esta parte que ya se presenta el programa terminado se implementaron dos nuevos operadores ariméticos, estos son: operaciones trigonométricas, logarítmicas. El objetivo principal del proyecto es poder ingresar una expresión que contengan valores en los distintos sistemas de comunicación y presentar la respuesta de la expresión en todos los sistemas de numeración nombrados anteriormente.

El sistema de definición de caracteres del presente proyecto es básico, esto quiere decir, que a los diferentes símbolos se los reconocerá como concepto de operador, número de algún sistema de numeración y símbolo de agrupación. Se construyó una estructura sólida en el reconocimiento de meta caracteres que permite reconocer cada elemento y descomponerlo en su mínima expresión para poder respetar todas las reglas aritméticas dentro de la expresión arimética. El lenguaje usado por defecto en LEX y YACC es C por lo que es indispensable tener un conocimiento previo sobre este lenguaje. Las librerías, funciones y lógicas que se utilizarán en este informe van a ser escrita únicamente ene l e lenguaje ya mencionado.

En este informe se detallará todas la teoría y la práctica utilizada para poder crear el programa. Por tanto, cada sección escrita está sumamente analizada comprobada y verificada por estudiantes de la Escuela Politécnica Nacional.

**Objetivos:**

* Observar y comprender el uso del analizador léxico de un compilador mediante el uso de los archivos.lex y yacc.
* Recordar los elementos fundamentales para el uso de expresiones regulares.
* Generar una calculadora que tenga como base el analizado léxico y el analizador sintáctico, utilizando los estándares ariméticos y numéricos de los distintos sistemas de numeración.
* Componer un programa práctico que pueda componerse de distintos archivos utilizando un código, una entrada y una Salida.
* Estructurar un código que pueda ser reconocido por el compilador de C (LEX y YACC) que sea comparable con programas hechos en otros lenguajes de programación.
* Desglosar una operación aritmética mixta e interpretar los elementos que la componen.
* Aprender sobre la estructura básica que debe tener un código para que se pueda ser ejecutado por medio de lex y yacc.

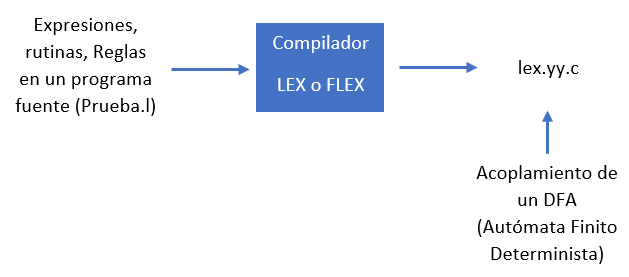
**Marco teórico:**

**Analizador Léxico:**

LEX es un generador de analizadores léxicos, es decir, ayuda a producir programas, para esto, se introducen expresiones regulares y las acciones o rutinas ligadas a dichas expresiones.

Lo antes mencionada, devuelve un programa en C, el cual se define por sintaxis de LEX y rutinas, todo esto definido por el usuario.

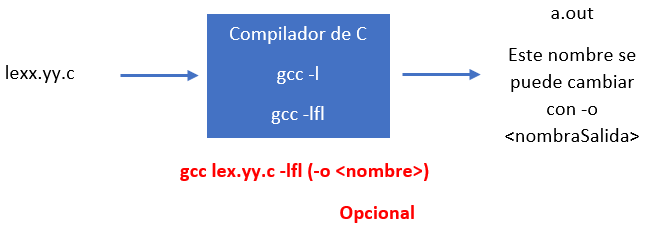
El reconocimiento léxico consiste en enlazar el flujo de entrada con el patrón de alguna expresión regular, esto retorna un lexema (token (componente léxico apropiado) apropiado y el valor asociado), esto se puede apreciar a continuación en la figura 1:



*Figura 1: Flujo de generación del analizador léxico*

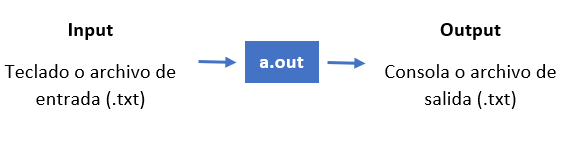
El código lex puede reconocer expresiones en una cadena de entrada y la divide en caracteres coincidentes con las expresiones, esto gracias a que tiene implementado un autómata.

GCC pertenece a una colección de compiladores estándar para varios sistemas operativos, se encarga de identificar distintos tipos de datos, copiarlos, traducirlos y generar un programa de salida (a.out) por defecto.



*Figura 2: Flujo de creación del programa de salida capaz de realizar las acciones y rutinas solicitadas.*

El programa a.out es el analizador léxico que se encarga de convertir una cadena de entrada en una secuencia de tokens. Este programa puede recibir como entrada una secuencia ingresada por teclado o en un archivo de entrada.

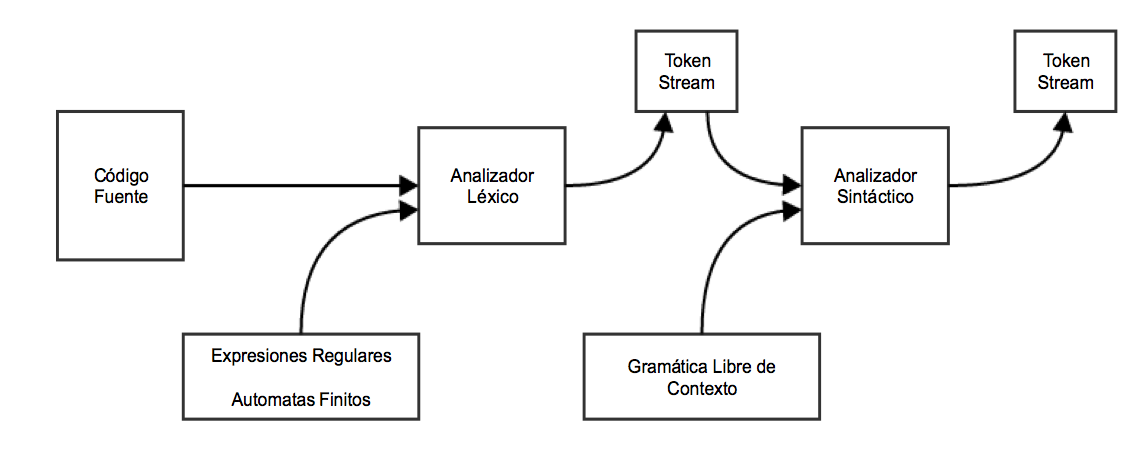


*Figura 3: Funcionamiento y relación de dispositivos de entrada y salida con a.out*

El redireccionamiento del modo entrada y salida se lo realiza con < y >, respectivamente.

**Analizador sintáctico:**

El analizador sintáctico de C o también llamado yacc es una derivación independiente de LEX, esto quiere decir que ha sido escrito usando la variante de FLEX como lenguaje principal. Yacc es un programa dentro del compilador de C que recibe las sentencias analizadas por lex y las trasforma en otro lenguaje para que así el analizador semántico pueda entender las sentencias deseadas por el programador.



*Figura 4: Entrada y salida de código de un analizador léxico y sintáctico.*

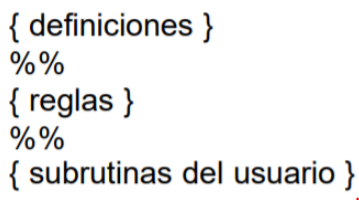
Un compilador se compone se distintos softwares que reciben y entregan código transformándolos en un metalenguaje que pueda entender el sistema que le precede. Tal como se muestra en la Figura 4.

Puesto que en analizador sintáctico no puede operar desde un código fuente directamente necesita de un intermediario que le traduzca. Para yacc este es el analizador léxico que trabaja conjuntamente para poder analizar las sentencias.

**Diseño del Programa:**

**Diseño en lex:**

Para la elaboración del proyecto se ha utilizado la estructura común de un programa lex:



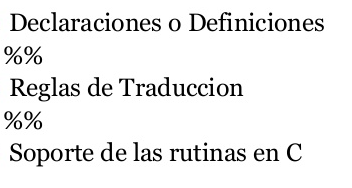
*Figura 5: Estructura básica de un programa lex.*

Colocando en la sección de definiciones las bibliotecas del lenguaje C que necesitamos para ejecutar ciertas instancias del código:

**Diseño en yacc:**

El generador de análisis sintáctico no es solo un analizador sino un generador de analizadores. A partir de un fichero yacc se genera un fichero en C que contiene un análisis sintáctico del programa. Yacc por sí solo no tiene funcionalidad, debe estar unido con un analizador léxico para funcionar. Cuando se genera el archivo C en este se van a encontrar llamadas yylex() que debe estar definida y devolver el tipo de lexema que encuentra. Además, a diferencia delos archivos lex, yacc debe tener obligatoriamente la función yyerror() por si lex encuentra una cadena no encaja en la gramática escrita.

Yacc al igual que lex presenta muchas similitudes ya que yacc fue programado utilizando lex. La estructura básica de un programa en yacc es la siguiente:



*Figura 6: Estructura básica de un programa en yacc.*

En la parte de declaraciones y definiciones, en comparación con lex que se definen las expresiones regulares, van la declaración de tokens que se van a usar en las terminales. Un ejemplo para esta sección podría ser:

*%token MUL*

*%token DIV*

*%token COS*

*Figura 7: Ejemplo de sección de definiciones*

A diferencia de lex, de estas tres secciones, solo la traducción de reglas es obligatoria y no debe estar vacía. Esta sección es donde se tiene que implementar las terminales. Un ejemplo en esta sección es:

*Expresion: TKN\_NUM {$$=$1;} |*

*Expresion TKN\_MAS Expresion {$$=$1+$3;}*

*Figura 8: Ejemplo de sección de reglas*

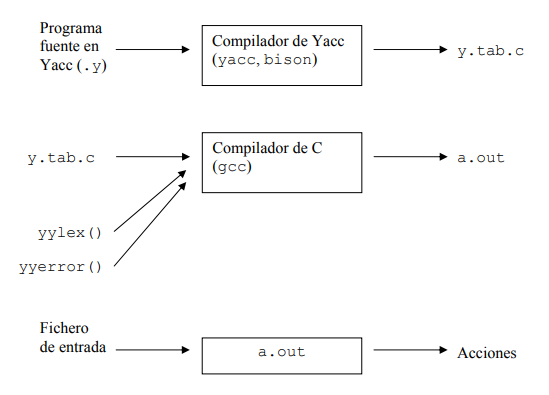
Considerando esto, concluiríamos que el mínimo programa en yacc es:

*%%*

*Regla gramatical acción*

*Figura 9: Ejemplo de sección de rutinas*

Dentro de la sección de rutinas es donde se escribirá el main() del programa. A diferencia de lex, que no era obligatorio escribir la función main(), en yacc es necesario escribir esta. Por tanto hay que incluir el yylex(), el yyerror() y el main() fuera o dentro del programa (si es externo el main hay que enlazarlo al archivo yacc).



*Figura 10: Componentes que participan al compilar un archivo yacc.*

Al igual que pasa con lex, al compilar yacc genera un a.out que conecta a los dos archivos lex y yacc. Cuando queremos ejecutar un archivo yacc lo primero que se necesita hacer es compilar el archivo lex. Una vez compilado lex seguiremos con la compilación de yacc. En este momento tenemos compilados los dos archivos, pero no presentan ninguna conexión. Luego se ingresa un comando que permite unir a los dos archivos y se compilan de manera unida. Finalmente se corre el a.out.

**Comandos para compilar y ejecutar los archivos lex y yacc.**

Todo lenguaje que depende de un compilador para ejecutarse necesita de un compilador externo. En el caso del analizador léxico y sintáctico su compilador predeterminado es el de C.

Para compilar un archivo lex usaremos el siguiente comando:

*$ lex nombreDelArchivo.l*

*Figura 11: Comando para compilar archivo lex*

Cuando queremos compilar un archio yacc usaremos el siguiente comando:

*$ yacc -d -v nombreDelArchivo.y*

*Figura 12: Comando para compilar archivo yacc*

Finalmente, para compilar un programa que una a yacc y a lex se debe usar:

*$ gcc lex.yy.c y.tab.c -lfl -lm*

*Figura 13: Comando para compilar archivo yacc y lex combinados (creación del a.out)*

Los pasos mostrados anteriormente son solo para ejecutar en conjunto los dos archivos lex y yacc. Si se desea únicamente ejecutar el archivo lex deberemos únicamente compilar el archivo lex y después ejecutar la siguiente sentencia para crear el a.out del archivo lex:

*$ gcc lex.yy.c -lfl*

*Figura 14: Comando para crear el a.out*

**Expresiones Regulares:**

Una expresión regular es una secuencia de caracteres que conforman un patrón de búsqueda. Principalmente son usadas para la búsqueda de tipos de cadenas u operaciones. Asimismo, son usadas para determinar una combinación de caracteres dentro de una cadena de texto. Las expresiones regulares permiten una muy flexible búsqueda o reconocimiento en un texto determinado.

La mayoría de las notaciones a las que hace referencia las expresiones regulares expresan autómatas finitos y se construyen utilizando caracteres del alfabeto sobre el cual se define el lenguaje. Por ejemplo, un patrón de expresiones regulares podría ser el patrón:

^[0-9]+”.”?[0-9]\*$

*Figura 15: Expresión regular que reconocer valores con o sin decimales.*

La expresión anterior es una combinación de letras del alfabeto con símbolos especiales, las cuales al ingresarlas en un analizador léxico e ingresar una cadena solo reconocerá aquellas líneas que comiencen con uno o más valor del 0 al 9, seguidos o no con un punto y luego precedidos o no por uno o más valores del 0 al 9.

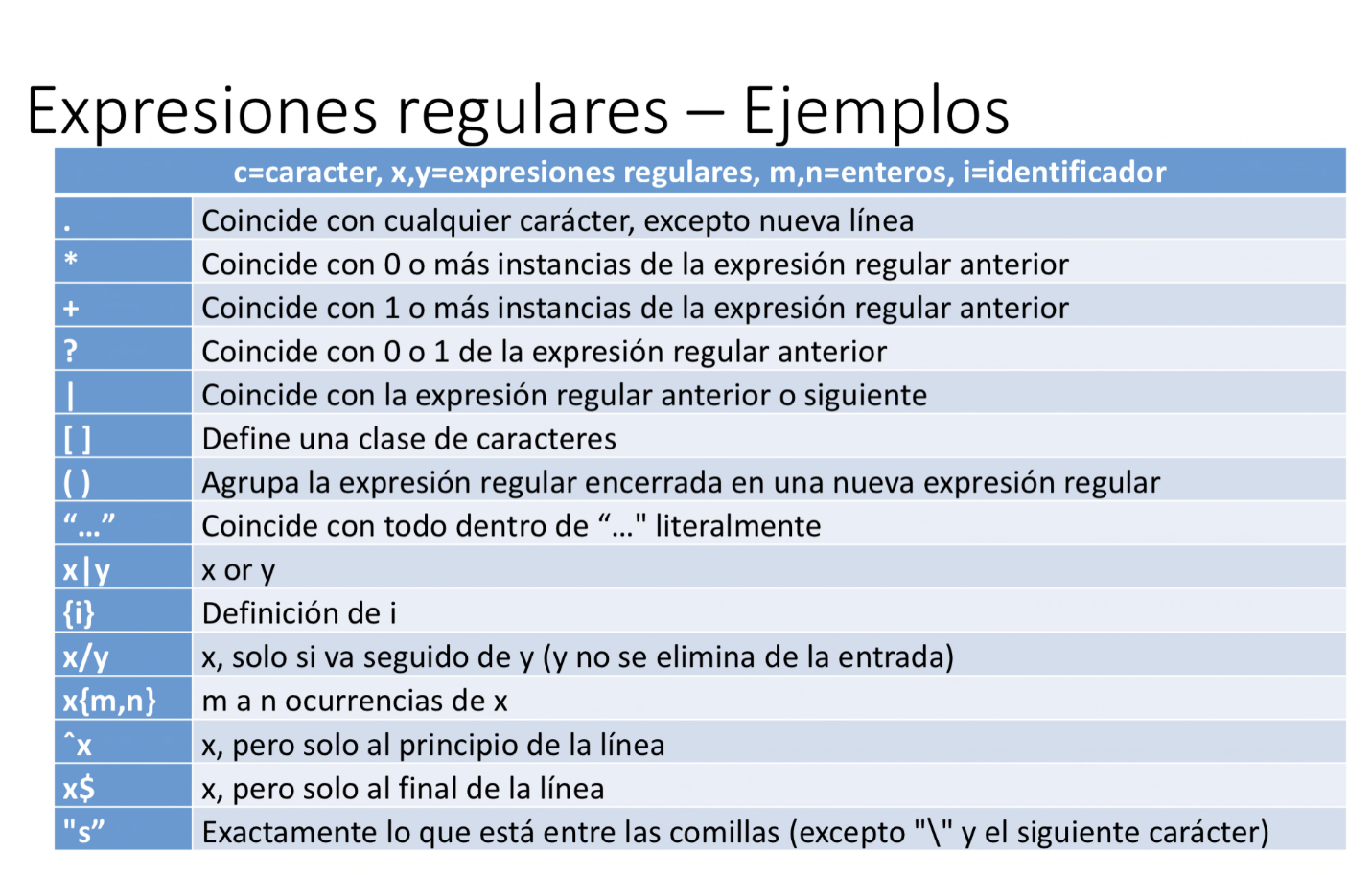
**Usos de expresiones regulares:**

El uso más común en para la expresión regales es describir un conjunto de cadenas para una determinada función, resultando de utilidad en editores de texto y aplicaciones para buscar y manipular texto. Otro uso adecuado es buscar y reemplazar patrones en un texto.

Actualmente las expresiones regulares facilitan a los programadores la detección de cadenas. Inicialmente este reconocimiento de cadenas se programa para cada aplicación sin mecanismo alguno inherente al lenguaje de programación, pero con el tiempo se ha venido incorporando el uso de las expresiones regulares.

**Símbolos para crear patrones de expresiones regulares**

Así como todo lenguaje de programación, las expresiones regulares deben de tener una estructura al ser escritos. En este caso cuando se comienza a escribir un patrón es indispensable escribir el símbolo ^ cuando se desea iniciar una cadena y finalizarla con $ (Figura 5). Esto hará que solo reconozca cadenas que entren en las restricciones escritas dentro del símbolo ^ y $. Algunas reglas gramaticales y símbolos que se usan por lo general son:



*Figura 16: Notaciones y símbolos frecuentes al escribir expresiones regulares.*

**Generador de analizadores léxicos (lex)**

El generador de analizador léxico como tal es la función yylex() que devuelve 0 cuando se ha alcanzado el fin de fichero y el resultado de llamar a yywrap() es 1. yylex() puede devolver otros valores si se incluye sentencias return como parte de una acción dentro de la sección de reglas.

Aparte de la función yylex() y yywrap() existen varias funciones que puede realizar el analizador léxico, entre ellas están:

* **yytext():** Es usada para imprimir la coincidencia que se encontró de acuerdo con la expresión regular.
* **yymore():** Esta función cuando se la llama indica a lex que la siguiente cadena a leer no se la colocará en la siguiente línea en el yytext, sino que lo haga a continuación.
* **yyles(int n):** Introduce en la entrada todos los caracteres de yytext, excepto los n primeros.
* **input():** Devuelve el siguiente carácter de la entrada, y avanza en 1 el punto de lex.
* **unput(int c):** Coloca en la entrada el carácter ‘c’ que será la siguiente que lea lex.
* **yywrap():** Esta función es ejecutada cuando lex llega al final del fichero. Tiene dos funciones: devuelve el control a la función a la que fue llamada (yywrap()=0), yylex() seguirá procesando la entrada como si no hubiera sucedida anda(yywrap=1).

**Análisis de las líneas de código en el archivo proyectob2.l**

***Sección de definiciones:***



*Figura 17: Sección de definiciones en el archivo proyectob2.l*

Sección donde definimos las bibliotecas de C con las cuales vamos a trabajar, algunas de las funciones que nos brindan dichas librerías son: printf() para mostrar elementos en pantalla (terminal), scanf() que sirve para leer entradas por el teclado, entre otras funciones que se detallaran más adelante.

También se implementan los nombres y argumentos de funciones que fueron creadas precisamente para este programa, estas pueden ser definidas en esta sección, pero para mantener una mejor estructura y legibilidad del código, se definieron en la sección de subrutinas de usuario, después de la función principal main().

A continuación, se caracterizan las librerías definidas para la realización del programa:

* **<stdio.h>:** Incluye funciones, constantes y macros para la salida e ingreso de datos, ejemplos de estos, son las funciones printf() y scanf(), además de otras funciones muy útiles para el manejo de información.
* **<string.h>:** Contiene varias funciones que ayudan en la manipulación de cadenas, es decir, arreglos de caracteres.
* **<ctype.h>:** Archivo de cabecera diseñado para transformar y clasificar caracteres, de manera individual.
* **<stdbool.h>:** Permite utilizar variables de tipo booleano (verdadero o falso).
* **<math.h>:** Este archivo de cabecera implementa varias operaciones matemáticas básicas, tales como radicación, exponenciación, entre otras, también incluye varias funciones trigonométricas, de parte entera (piso, techo, redondeo), etc.
* **“y.tab.h”:** contiene la codificación de los símbolos terminales de la gramática.

En la sección de definiciones, también se crean identificadores para expresiones regulares, esto no es obligatorio, pero ayudan a comprender de mejor manera el funcionamiento del programa.

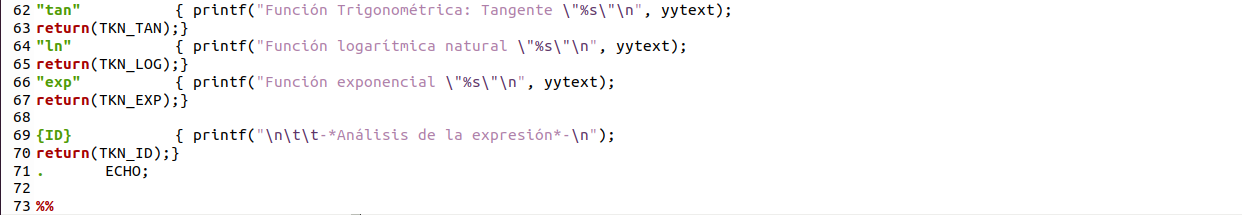
Los identificadores creados fueron los siguientes:

* **WS (White space)**: El cual define uno o varios espacios en blanco, los cuales pueden provenir de la tecla tabulador o espacio.
* **agrupacionA**: que se define con todos los diferentes tipos de signos de agrupación de apertura.
* **agrupacionC**: que se define con todos los diferentes tipos de signos de agrupación de cierre.
* **DIGITO**: es una de las expresiones más básicas que se puede definir con expresiones regulares, pero se puede aplicar para definir otras mas complejas tales como números decimales o enteros.
* **Romano**: expresión regular para validar números enteros en sistema de numeración romano, para este caso, no se identifican ciertas reglas del sistema romano, además, se estableció que P sea equivalente a 5000 y T a 10000 debido a las limitaciones léxicas propias del entorno.
* **binario**: expresión regular para validar número binarios, para diferenciarlo de los demás sistemas de numeración, se estableció que estos deben empezar con “b” o “B” de binario.
* **octal**: expresión regular para validar números octales, al igual que en el caso de los números binarios, se estableció que empiecen con “o” u “O” de octal.
* **hexadecimal**: números en sistema hexadecimal, en el caso de los números del 10 al 15, dado que estos se escriben con las letras de a la A la F, no importa que estas estén en minúsculas o mayúsculas. Además, se diferencia de los demás sistemas de numeración, estableciendo que empiece con “h” o “H” de hexadecimal.
* **ID:** variable a la cual se le asigna el valor de la expresión aritmética mixta.

***Sección de reglas:***

En esta sección se define tanto los signos que pueden aparecer dentro de nuestra ecuación, las posibles cadenas a ser reconocidas como números y los operandos que se pueden utilizar. Además, con ayuda de la función printf podemos imprimir en la terminal el tipo al que pertenece dicha cadena dentro de la ecuación. Cabe recalcar que se definió que la palabra “sqrt” haga referencia a la raíz cuadrada.

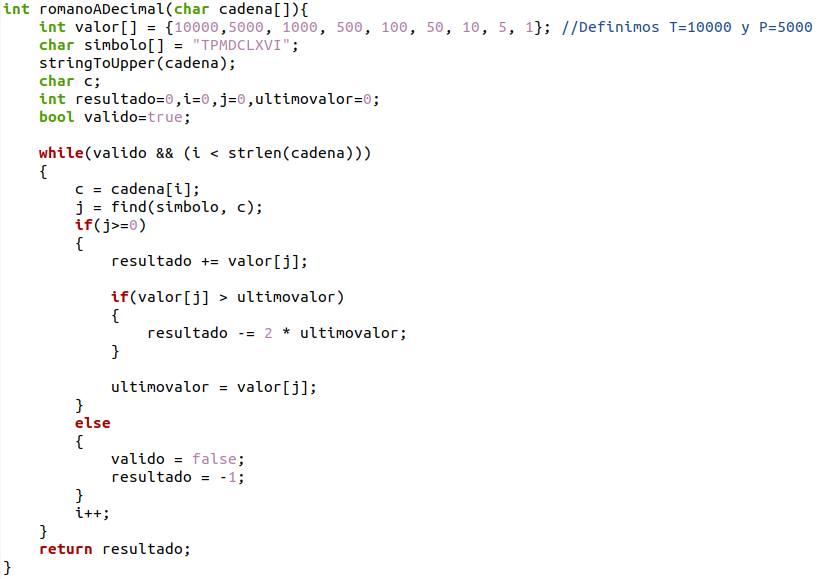




*Figura 18: Sección de reglas del programa en el archivo proyectob2.l*

También se definen las acciones ligadas al encuentro con los números expresados en los distintos sistemas de numeración, por ejemplo, si se encuentra un numero romano, este es identificado como tal y además es transformado en su equivalente en decimal. Dicha acción se realiza con todos los demás sistemas de numeración, esta transformación es gracias a ciertas funciones las cuales fueron definidas en la sección de definiciones e implementadas en la sección de subrutinas usuario. Estos valores son asignados a variables propias de flex para que puedan realizarse el proceso matemático en el archivo proyectob2.y, además se devuelve el token correspondiente para cada secuencia de caracteres que se ubiquen.

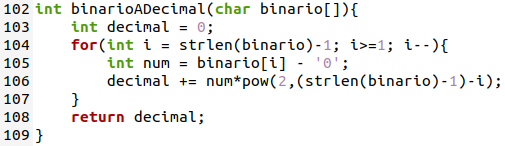
***Sección de subrutinas:***



*Figura 19: Función romanoADecimal() en la sección de subrutinas*

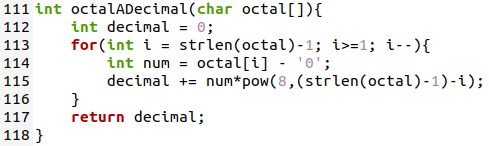
Como su nombre indica, esta función, transforma un numero romano a su equivalente en decimal, para ello, se le pasa por argumento un arreglo de caracteres (cadena en C), y se retorna el correspondiente entero. Es necesario aclara que la función no determina si el número es válido, de esto se encarga lex, el cual tampoco puede determinar que el numero romano cumpla con todas las reglas establecidas para la notación romana de números. Aun así, el programa devuelve el correspondiente valor del número romano, cumpla este o no las reglas.

Para esta función y la que convierte de hexadecimal a decimal, se utilizó la función find(), la cual se explicara su funcionamiento más adelante.



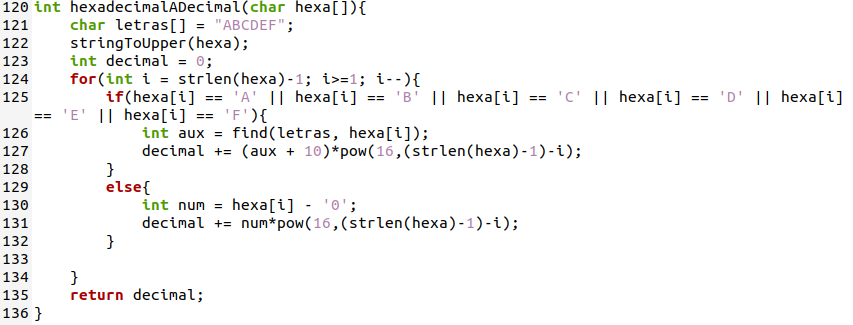
*Figura 20: Función binarioADecimal() en la sección de subrutinas*

Función que toma como argumento un numero binario, este debe empezar con “b” o “B” para poder diferenciarlos de los demás sistemas, es decir, el argumento es un arreglo de caracteres, y se devuelve un valor entero correspondiente a su equivalencia en binaria.



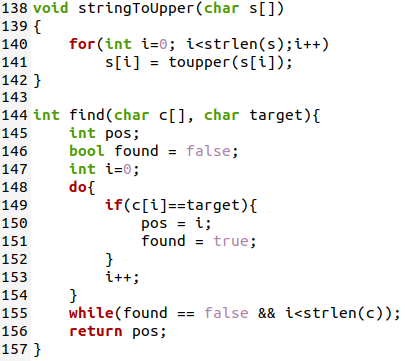
*Figura 21: Función octalADecimal() en la sección de subrutinas*

Función que funciona de manera similar a la anterior, toma como argumento una cadena de caracteres que cumple con las condiciones de ser un octal, para posteriormente retornar el entero decimal correspondiente a dicha cadena.



*Figura 22: Función hexadecimalADecimal() en la sección de subrutinas.*

Para esta función se utiliza la función stringToUpper() la cual transforma en mayúsculas todos los caracteres de una cadena, esta para evitar una programación por casos y solo centrarse en un caso en general. Igualmente, esta función recibe como argumento una cadena de caracteres correspondiente a un número hexadecimal y devuelve un entero decimal acorde al valor de este.



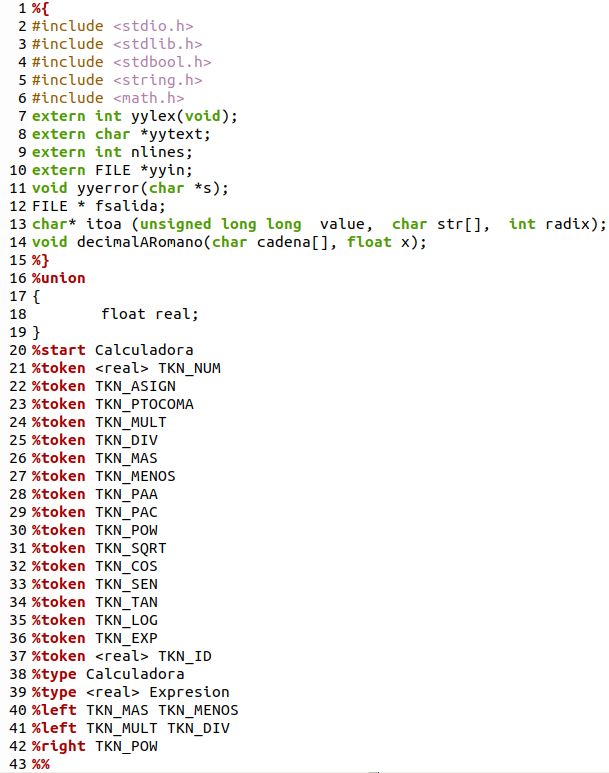
*Figura 23: Funciones stringToUpper() y find() en la sección de subrutinas*

Este par de funciones son complementarias, es decir, las demás antes mencionadas usan de estas para llevar a cabo su función. StrinToUpper() transforma cada uno de los caracteres de una cadena en mayúsculas, para esto se le pasa como argumento la cadena, y la función find() recibe dos argumentos, una cadena, y un carácter, su objetivo es buscar el carácter en la cadena y retornar la posición de la primera ocurrencia del carácter en la cadena.

Todas estas funciones son fáciles de implementar y van a servir de mucha ayuda en la siguiente parte de este proyecto, donde se realizarán operaciones con cada uno de los números en los diferentes sistemas de numeración.

**Análisis de las líneas de código en el archivo proyectob2.y**

***Sección de definiciones:***



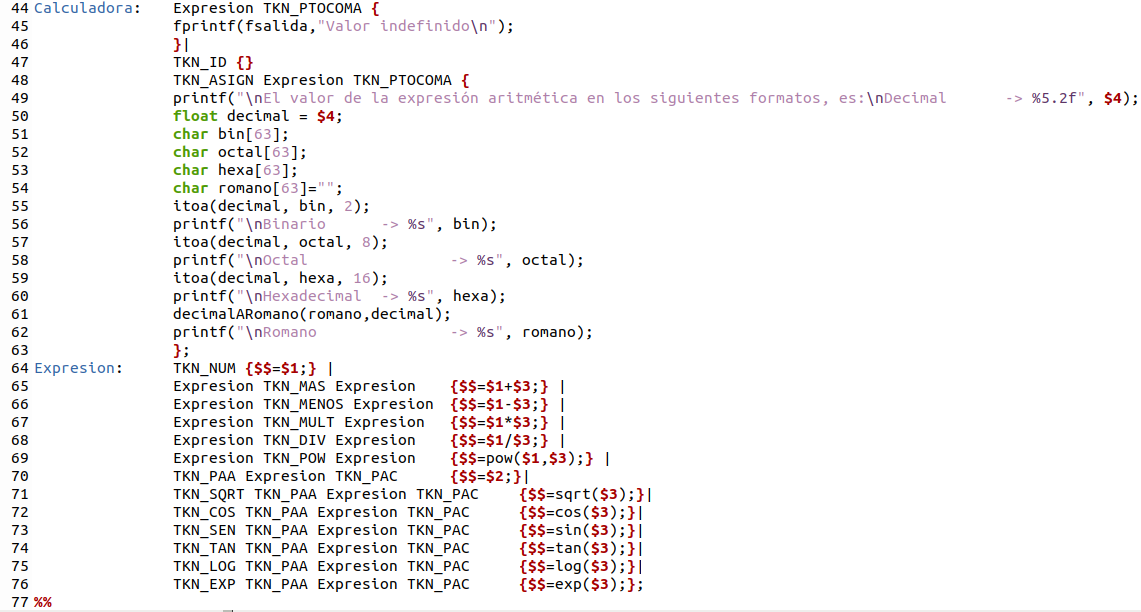
*Figura 24: Sección de definiciones en el archivo proyectob2.y*

Se incluyen los archivos y librerías necesarias para que el programa cumpla con su objetivo, a su vez, se declaran variables externas y las definiciones de las funciones necesarias para transformar la respuesta en decimal a los demás formatos permitidos por la calculadora multiformato. Las estructuras unión también son declaradas aquí, así como, todos los símbolos de terminal y la regla de inicio predeterminada para la sección de reglas, la cual por defecto es la primera si no se especifica.

Además, con el objetivo de que la respuesta sea la adecuada pese a que se trabaje con una gramática ambigua, se declararon reglas de precedencia para operaciones como: suma, resta, multiplicación, división y potencia, las cuales requieres de dichas reglas.

***Sección de reglas:***

De forma parecida que en el archivo proyectob2.l, aquí se declaran las acciones que se realizan al analizar el flujo de entrada, para ello se definen la gramática y las rutinas semánticas asociadas, esto con el fin de realizar las operaciones necesarias para llegar a una respuesta.

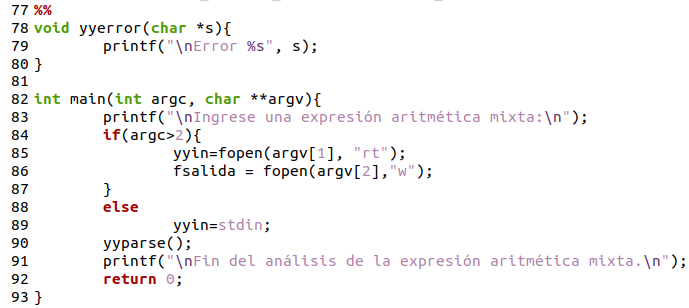


*Figura 25: Sección de reglas del programa en el archivo proyectob2.y*

Se hace uso de los tokens que se retornaban en el archivo .lex y los que fueron declarados adicionalmente en la sección de definiciones, gracias a estos tokens y a las rutinas semánticas se logra obtener la respuesta de la expresión mixta introducida y devolver dicha respuesta en todos los sistemas de numeración permitidos por esta calculadora.

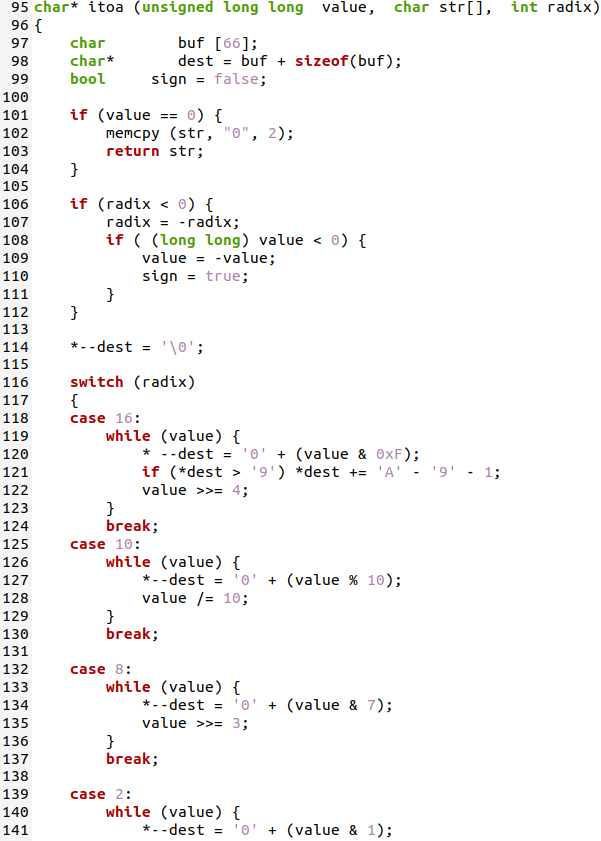
***Sección de subrutinas:***

Tal como el nombre lo indica, aquí se declaran las rutinas de usuario necesarias para que el programa pueda ejecutarse manera correcta.

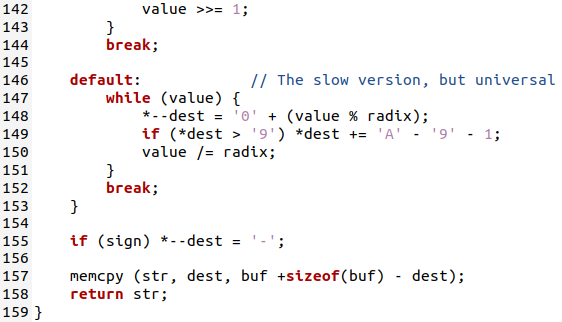


*Figura 26: Función yyerror() y main() en la sección de subrutinas del archivo proyectob2.y*

La función yyerror() es llamada automáticamente si se encuentra algún inconveniente al momento de analizar el flujo de entrada, es una función propia de yacc y su estructura puede ser modificada de acuerdo a los requerimientos del usuario. En la función main() se establecen las acciones principales a realizarse y también se invoca al parse de yacc para que comience con el proceso de análisis.

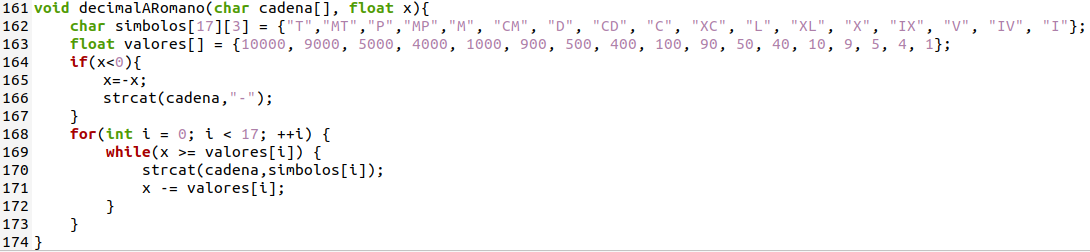


*Figura 27: Función itoa() en la sección de subrutinas del archivo proyectob2.y*



*Figura 28: Función itoa() en la sección de subrutinas del archivo proyectob2.y*

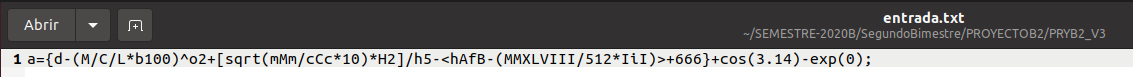
La función itoa() es propia de la librería <stdlib.h> al igual que otras funciones tales como atoi(), atof(), entre otras. Sin embargo, en el sistema operativo Linux no se encuentra definida por defecto así que esta fue declarada y utilizada para convertir un numero decimal, almacenado en una variable float, a los demás sistemas de numeración, esta función es muy útil, puesto que solo se debe pasar por parámetro el número, un array de caracteres (string) y el formato a transformar, y esta función colocara la cadena en la cadena pasada como argumento.



*Figura 29: Función decimalARomano() en la sección de subrutinas del archivo proyectob2.y*

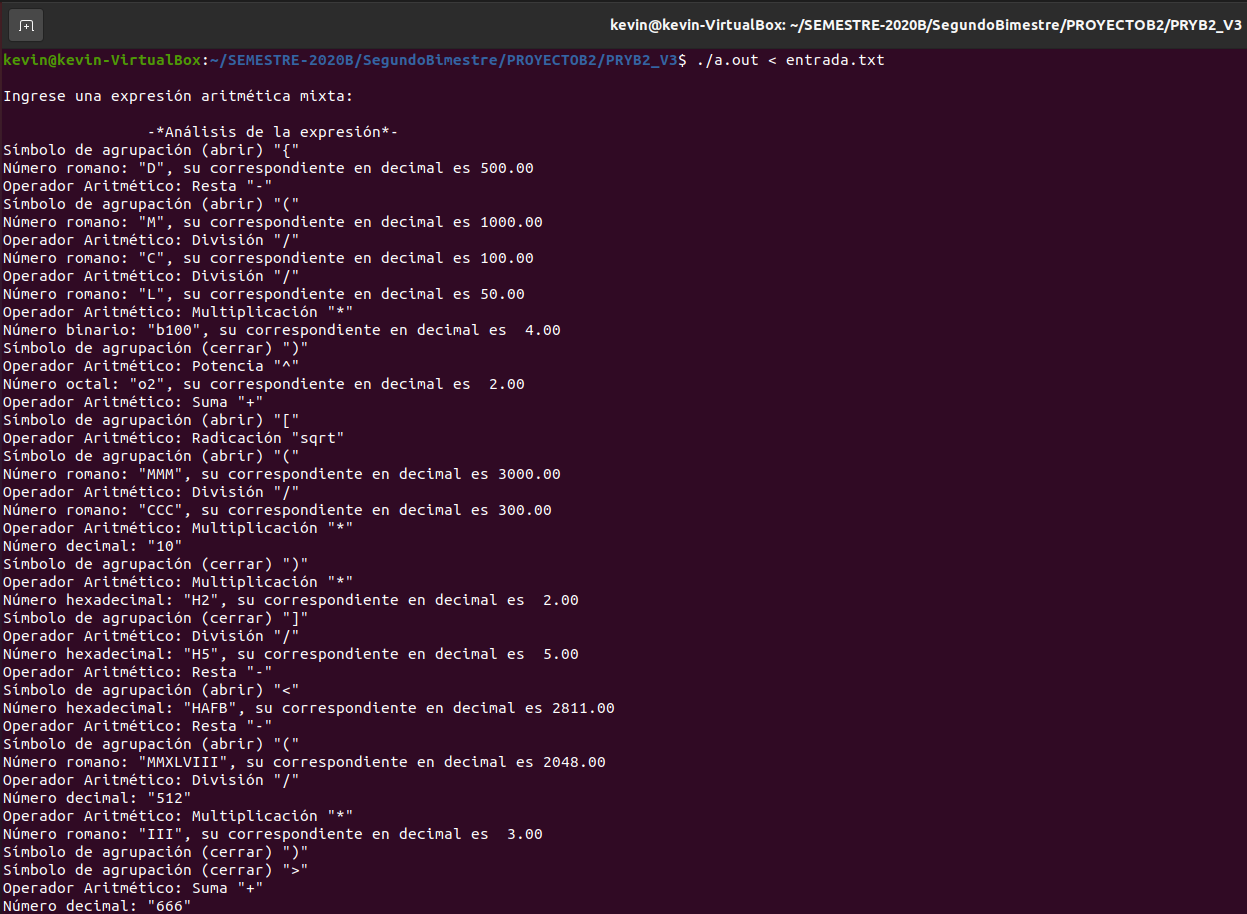
Función que devuelve la equivalencia en sistema romano de un numero decimal que se le pasa por parámetro, para ello, también es necesario pasar un arreglo de caracteres el cual será modificado con el correspondiente número romano. Esta función realiza el procedimiento inverso al visto en la función romanoADecimal() en el archivo proyectob2.l, pero la lógica a seguir es muy parecida.

Para realizar las pruebas de escritorio se usó la siguiente expresión mixta:

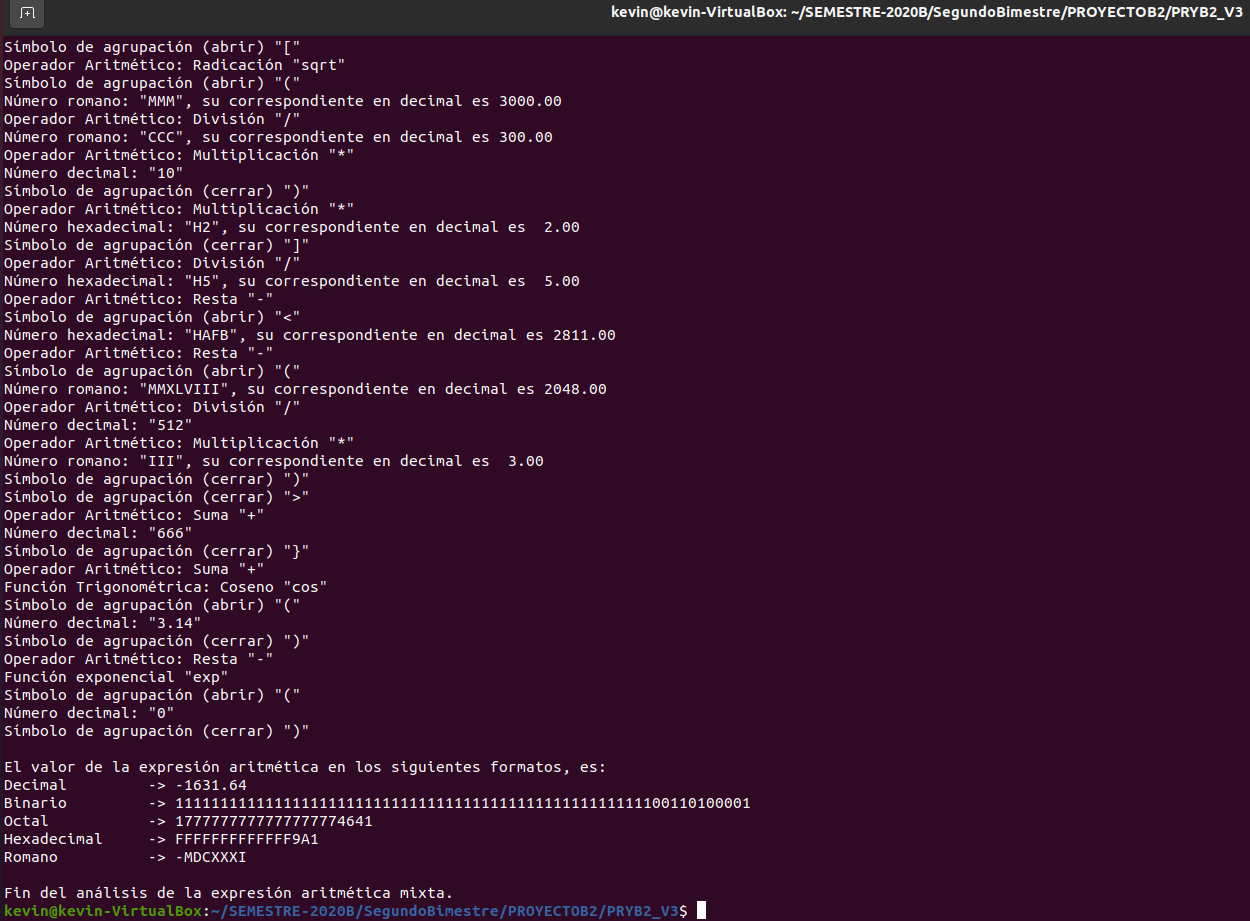


*Figura 30: Operación aritmética mixta a analizar*

Para esto se compila el ejecutable a.out el cual recibe como entrada un archivo .txt con dicha expresión (entrada.txt)



*Figura 31: Ejecución del a.out y como entrada el archivo entrada.txt*



*Figura 32: Ejecución del a.out y como entrada el archivo entrada.txt*

El programa analiza cada uno de los elementos del archivo de entrada, o en su defecto, del flujo ingresado por consola, los números en otros sistemas de numeración que no sean decimal, son transformados a su correspondiente en decimal y esto sirve de ayuda para calcular la respuesta de la expresión. Este resultado es obtenido en formato decimal y este valor es transformado a los demás sistemas de numeración, cabe mencionar que solo es transformada la parte entera mas no la decimal, además, los números negativos en los sistemas de numeración binario, octal y hexadecimal son representados en formato complemento a 2.

**Conclusiones y recomendaciones**

* El uso del analizador léxico mediante los archivos .lex son muy útiles para validar una cadena. Por lo general son utilizados para verificar correos electrónicos, direcciones, operaciones, etc.
* Lex o Flex es una parte del compilador que recibe expresiones regulares y lo expresa en archivos lex.yy.c. Esta parte del compilador es importante ya que trasforma el lenguaje de alto nivel a un lenguaje en el que las demás partes del compilador puedan entender.
* El código lex reconoce líneas con expresiones regulares y estas las divide en caracteres que pueden o no coincidir con una cadena preferencial. Esto lo hace gracias a que lex tiene implementado un autómata de reconocimiento de expresiones regulares.
* El compilador de C es el que recibe los archivos lex.yy.c y los convierte en archivos ejecutables para Ubuntu.
* Es necesario tener un conocimiento previo sobre la estructura básica de los archivos .lex ya que para construir el programa existen diferentes apartados para incluir código, librerías y expresiones regulares.
* Utilizar el comando **gcc lex.yy.c -lfl- lm** para generar el ejecutable del programa, este último es necesario dado que se utiliza la librería <math.h> en el código.

**Bibliografía:**

* Chancusig R. (2020) “Sesión 1 – 18 de noviembre-2020" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 4 – 9 de diciembre-2020" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 7 – 6 de enero-2021" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 11–3-febrero-2021" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 12–10-febrero-2021" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 13–17-febrero-2021" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 14–24-febrero-2021" [Diapositiva de Power Point].
* Chancusig R. (2020) “Sesión 15–3-marzo-2021" [Diapositiva de Power Point].
* Simmross F., Universidad de Valladolid (S.F.). El generador de analizadores léxicos lex. Recuperado el 17 de Enero de 2021, disponible en: <https://www.infor.uva.es/~mluisa/talf/docs/labo/L6.pdf>
* Simmross F, Universidad de Valladolid (S.F.). El generador de analizadires sintácticos yacc. Recuperado el 9 de Marzo de 2021, disponible en: <https://www.infor.uva.es/~mluisa/talf/docs/labo/L8.pdf>
* Gómez S. Guía para entender y usar expresiones regulares. (Enero 29, 2020). Recuperado el 17 de Enero de 2021, disponible en: <https://www.adictosaltrabajo.com/2015/01/29/regexsam/>