Análisis Lexicográfico y Sintáctico

Teoría de la Computación

John Ochoa Abad

00345743

13 de diciembre de 2025

1. Introducción

La Teoría de la Computación proporciona los fundamentos formales para el estudio de los lenguajes, los autómatas y los modelos de cómputo. Dentro de este marco teórico, el análisis lexicográfico y el análisis sintáctico desempeñan un papel esencial en la comprensión y el procesamiento de los lenguajes formales, especialmente en el contexto del diseño e implementación de compiladores e intérpretes.

El análisis lexicográfico constituye la primera fase del proceso de traducción de un lenguaje, y se encarga de transformar una secuencia de caracteres de entrada en una secuencia de unidades significativas denominadas *tokens*. Este proceso se basa en la teoría de los lenguajes regulares y en el uso de autómatas finitos, los cuales permiten reconocer patrones definidos mediante expresiones regulares de manera eficiente (Aho & Ullman,2007).

Por su parte, el análisis sintáctico tiene como objetivo verificar si la secuencia de tokens generada por el analizador léxico cumple con las reglas gramaticales del lenguaje. Estas reglas suelen describirse mediante gramáticas libres de contexto, y el análisis se lleva a cabo utilizando estructuras como árboles de derivación o árboles sintácticos. Esta fase es fundamental para garantizar que los programas o expresiones estén correctamente estructurados antes de proceder a etapas posteriores del procesamiento (Sipser, 2012).

En conjunto, el análisis lexicográfico y sintáctico no solo son componentes clave en la construcción de compiladores, sino que también representan aplicaciones directas de conceptos centrales de la Teoría de la Computación, tales como autómatas, gramáticas formales y lenguajes, consolidando así la relación entre teoría y práctica en el estudio de los sistemas computacionales (Aho & Ullman, 2007).

2. Herramientas utilizadas

Para el desarrollo del presente proyecto se emplearon diversas herramientas clásicas y modernas utilizadas en la construcción de compiladores y analizadores de lenguajes formales. En particular, se utilizaron Lex y Yacc para la implementación de los analizadores lexicográfico y sintáctico, respectivamente, así como los lenguajes de programación C y Rust para la integración, ejecución y extensión de las calculadoras desarrolladas.

2.1. Lex

Lex es una herramienta generadora de analizadores lexicográficos que permite definir patrones léxicos mediante expresiones regulares y asociarlos con acciones escritas en un lenguaje de programación anfitrión. A partir de estas especificaciones, Lex genera automáticamente un programa en C que reconoce tokens en una secuencia de entrada. En el contexto de este proyecto, Lex fue utilizado para identificar los componentes básicos de las expresiones aritméticas, tales como números, operadores, paréntesis y símbolos especiales, tanto para números enteros y flotantes como para representaciones binarias y romanas. El uso de Lex permite aplicar directamente conceptos de lenguajes regulares y autómatas finitos estudiados en la Teoría de la Computación. Todo con el fin de poder obtener los tokens correctos para que funcionen de manera adecuada cada una de las calculadoras desarrolladas (Levine, Mason & Brown, 2019).

2.2. Yacc

Yacc es una herramienta destinada a la generación de analizadores sintácticos a partir de una gramática libre de contexto. Mediante la definición de reglas gramaticales y acciones semánticas, Yacc construye un parser capaz de verificar la estructura sintáctica de una secuencia de tokens producida por Lex. En este proyecto, Yacc fue empleado para definir las gramáticas de las calculadoras desarrolladas, asegurando el correcto manejo de la precedencia y asociatividad de los operadores aritméticos, así como el uso adecuado de paréntesis. De esta manera, Yacc permitió construir árboles sintácticos implícitos y realizar la evaluación correcta de las expresiones ingresadas por el usuario en cada una de las 3 calculadoras (Levine, Mason & Brown, 2019).

2.3. Lenguaje C

El lenguaje de programación C fue utilizado como lenguaje anfitrión para la integración de los analizadores generados por Lex y Yacc. Debido a su cercanía con el hardware y su eficiencia, C es tradicionalmente empleado en la implementación de compiladores e intérpretes. En este proyecto, C permitió definir las estructuras de datos, las funciones

auxiliares y las acciones semánticas necesarias para realizar los cálculos aritméticos correspondientes a cada tipo de calculadora. Además, su uso facilitó la compilación y ejecución eficiente de los programas generados por las herramientas Lex y Yacc.

2.4. Lenguaje Rust

Como ejercicio adicional, se exploró la implementación de un analizador lexicográfico y sintáctico utilizando el lenguaje Rust. Rust es un lenguaje de programación moderno que ofrece seguridad de memoria sin necesidad de recolección de basura, lo cual lo hace adecuado para el desarrollo de software robusto y eficiente. En el contexto de este proyecto, Rust permitió analizar cómo los conceptos de análisis lexicográfico y sintáctico pueden implementarse sin el uso directo de herramientas clásicas como Lex y Yacc, empleando en su lugar estructuras de control, expresiones regulares y técnicas de parsing propias del lenguaje (Rust Compiler Team, 2020). Esta implementación adicional permitió comparar enfoques tradicionales y modernos en la construcción de analizadores, reforzando la comprensión teórica y práctica del tema.

3. Solución de Ejercicios

3.1. Calculadora de Números

En esta sección se describe el desarrollo de una calculadora aritmética capaz de evaluar expresiones numéricas que incluyen números enteros y flotantes, operadores binarios y el uso de paréntesis. La implementación se realizó utilizando las herramientas Lex y Yacc, separando claramente las responsabilidades del análisis lexicográfico y del análisis sintáctico.

3.1.1. Análisis Lexicográfico (Lex)

El analizador lexicográfico fue desarrollado utilizando Lex y tiene como objetivo reconocer los distintos componentes léxicos presentes en las expresiones aritméticas de entrada. Entre los tokens identificados se encuentran los números enteros y flotantes, los operadores aritméticos básicos (`+`, `-`, `*`, `/`), los paréntesis y el carácter de nueva línea que indica el final de una expresión. Para el reconocimiento de números, se definieron expresiones regulares que permiten identificar tanto números enteros como números de punto flotante, incluyendo notación científica. Cada número reconocido es convertido a un valor de tipo `float` y almacenado en la estructura semántica compartida con Yacc, permitiendo su posterior evaluación en el análisis sintáctico.

Asimismo, el analizador lexicográfico ignora espacios en blanco y caracteres irrelevantes, garantizando que únicamente los tokens válidos sean enviados al parser. Esta

separación permite una entrada limpia y bien estructurada para la fase sintáctica.

3.1.2. Código del Analizador Lexicográfico

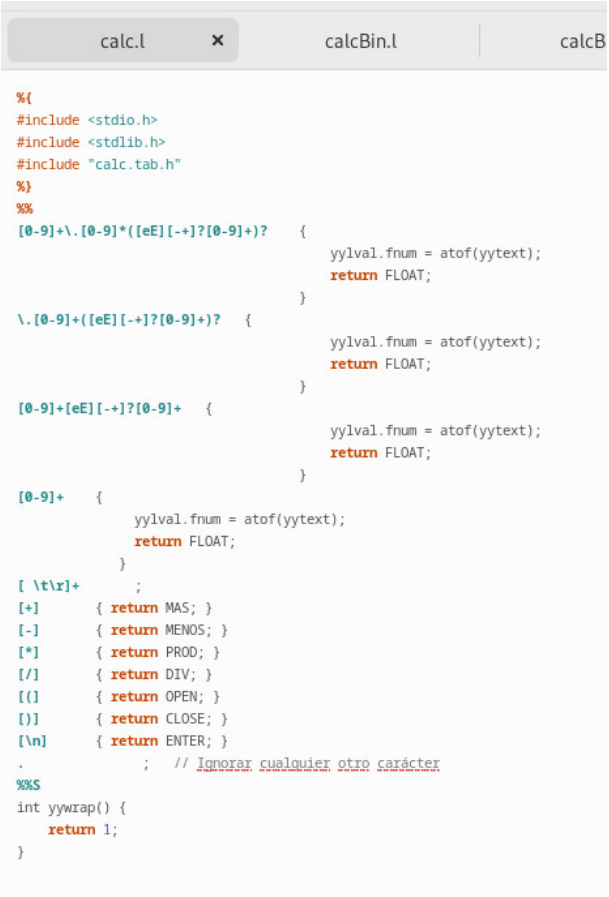

Figura 1: Código de Lex para Calculadora de Números

3.1.3. Análisis Sintáctico (Yacc)

El análisis sintáctico fue implementado utilizando Yacc, definiendo una gramática libre de contexto que describe la estructura válida de las expresiones aritméticas. La gramática permite el uso de operadores binarios de suma, resta, multiplicación y división, así como el uso de paréntesis para alterar la precedencia natural de las operaciones. Además, se incluye el manejo del operador unario negativo.

La precedencia de los operadores se establecieron lógicamente, asegurando que las operaciones de multiplicación y división tengan mayor prioridad que la suma y la resta. Esta definición permite evaluar correctamente expresiones complejas sin ambigüedades sintácticas.

Durante el proceso de análisis, las acciones semánticas asociadas a cada regla gramatical realizan directamente la evaluación de las expresiones, produciendo como resultado un valor numérico de tipo `float`. Adicionalmente, se implementó un manejo básico de errores sintácticos y semánticos, como la detección de divisiones entre cero, garantizando

una ejecución más robusta del programa.

3.1.4. Código del Analizador Sintáctico

calc.y × calc.l ca

```
%{
#include <stdio.h>
#include <stdlib.h>
extern int yylex(void);
extern void yyerror(char *s);
%}
%union {
    float fnum;
}
%token <fnum> FLOAT
%token MAS MENOS PROD DIV OPEN CLOSE ENTER
%type <fnum> expr
%left MAS MENOS
%left PROD DIV
%start input
%%
input:
    | input expr ENTER { printf("Resultado: %f\n\n", $2); }
    | input error ENTER { yyerror("Syntax error"); }
    ;
expr:
      FLOAT { $$ = $1; printf("Numero: %f\n", $$); }
    | MENOS expr { $$ = -$2; }
    | OPEN expr CLOSE { $$ = $2; }
    | expr PROD expr { $$ = $1 * $3; }
    | expr DIV expr {
        if ($3 == 0) {
            yyerror("Division entre cero");
            YYERROR;
        } else {
            $$ = $1 / $3;
        }
      }
    | expr MAS expr { $$ = $1 + $3; }
    | expr MENOS expr { $$ = $1 - $3; }
    ;
%%
int main() {
    printf("Ingrese una expresion: ");
    yyparse();
    return 0;
}
void yyerror(char *s) {
    printf("\nError: %s\n", s);
```

Figura 2: Código de Yacc para Calculadora de Números

3.1.5. Gramática

La gramática utilizada para la calculadora de números se define como una gramática libre de contexto que permite la evaluación de expresiones aritméticas con números enteros y de punto flotante, operadores binarios y el uso de paréntesis. La gramática es la siguiente:

$$
\begin{aligned}
\langle input \rangle &\rightarrow \langle input \rangle \ \langle expr \rangle \ \texttt{ENTER} \mid \epsilon \\
\langle expr \rangle &\rightarrow \texttt{FLOAT} \\
&\mid \texttt{MENOS} \ \langle expr \rangle \\
&\mid \texttt{OPEN} \ \langle expr \rangle \ \texttt{CLOSE} \\
&\mid \langle expr \rangle \ \texttt{PROD} \ \langle expr \rangle \\
&\mid \langle expr \rangle \ \texttt{DIV} \ \langle expr \rangle \\
&\mid \langle expr \rangle \ \texttt{MAS} \ \langle expr \rangle \\
&\mid \langle expr \rangle \ \texttt{MENOS} \ \langle expr \rangle
\end{aligned}
$$

3.1.6. Pruebas Realizadas

Para validar el correcto funcionamiento de la calculadora, se realizaron diversas pruebas utilizando expresiones aritméticas simples y compuestas. Estas pruebas incluyen el uso de paréntesis, operadores con distinta precedencia, números negativos y combinaciones de números enteros y flotantes. Los resultados obtenidos coinciden con los valores esperados, demostrando la correcta integración entre el análisis lexicográfico y sintáctico. Para los 3 casos de pruebas se mostro operaciones simple de resta, operación con paréntesis y el uso de todos los operadores y operación con manejo de números negativos

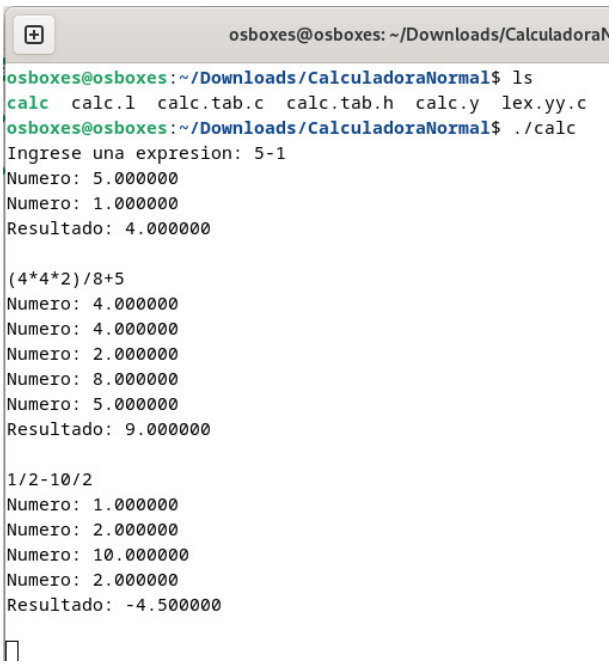

Figura 3: Pruebas realizadas para calculadora de números

3.2. Calculadora de Números Binarios

En esta sección se describe el desarrollo de una calculadora aritmética capaz de evaluar expresiones compuestas por números binarios, tanto enteros como con parte fraccionaria. La calculadora permite realizar operaciones aritméticas básicas utilizando la representación binaria, haciendo uso de operadores binarios y paréntesis. La implementación se realizó utilizando las herramientas Lex y Yacc, separando claramente las fases de análisis lexicográfico y sintáctico.

3.2.1. Análisis Lexicográfico (Lex)

El analizador lexicográfico fue desarrollado utilizando Lex y tiene como objetivo reconocer los distintos tokens presentes en las expresiones binarias de entrada. Entre los tokens identificados se encuentran los números binarios, formados por secuencias de dígitos `0` y `1`, con la posibilidad de incluir una parte fraccionaria mediante el uso del punto decimal, así como los operadores aritméticos básicos (`+`, `-`, `*`, `/`), los paréntesis y el carácter de nueva línea que indica el final de una expresión.

Para el reconocimiento de números binarios, se definieron expresiones regulares que permiten identificar correctamente valores binarios enteros y fraccionarios. Cada número reconocido es convertido internamente a un valor de tipo `float` mediante una función auxiliar, lo que facilita la evaluación de las operaciones aritméticas durante el análisis sintáctico. El analizador lexicográfico también ignora espacios en blanco y caracteres no relevantes, asegurando que únicamente los tokens válidos sean enviados al parser.

3.2.2. Código del Analizador Lexicográfico

3.2.3. Análisis Sintáctico (Yacc)

El análisis sintáctico fue implementado utilizando Yacc, definiendo una gramática libre de contexto que describe la estructura válida de las expresiones aritméticas binarias. La gramática permite el uso de operadores binarios de suma, resta, multiplicación y división, así como el uso de paréntesis para modificar la precedencia de las operaciones.

La precedencia de los operadores fue definida de manera que la multiplicación y la división tengan mayor prioridad que la suma y la resta. Aunque la entrada se realiza en formato binario, la evaluación de las expresiones se efectúa utilizando valores de punto flotante. Posteriormente, los resultados son convertidos nuevamente a su representación binaria para ser mostrados al usuario. Además, se implementó el manejo de errores semánticos, como la detección de divisiones entre cero, garantizando una ejecución segura del programa.

calcBin.l × | calc.y | calc.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "calcBin.tab.h"
float bin_to_float(const char *s) {
    float resultado = 0.0f;
    int punto_decimal = 0;
    float factor = 0.5f;
    for (const char *p = s; *p; p++) {
        if (*p == '.') {
            punto_decimal = 1;
            continue;
        }
        if (*p != '0' && *p != '1') continue;
        if (!punto_decimal) {
            resultado = resultado * 2 + (*p - '0');
        } else {
            resultado += (*p - '0') * factor;
            factor /= 2.0f;
        }
    }
    return resultado;
}
%}
```

Figura 4: Código de Lex para Calculadora de Números Binarios-Parte 1

calcBin.l × | calc.y | calc.l

```
%%
[01]+\.[01]*    {
    yylval.numBin = bin_to_float(yytext);
    return BINARIO;
}
\.[01]+         {
    yylval.numBin = bin_to_float(yytext);
    return BINARIO;
}
[01]+           {
    yylval.numBin = bin_to_float(yytext);
    return BINARIO;
}
[ \t\r]+        ;
[+]             { return MAS; }
[-]             { return MENOS; }
[*]             { return PROD; }
[/]             { return DIV; }
[(]             { return OPEN; }
[)]             { return CLOSE; }
[\n]            { return ENTER; }
.               ;   // Ignorar cualquier otro carácter
%%
int yywrap() {
    return 1;
}
```

Figura 5: Código de Lex para Calculadora de Números Binarios-Parte 2

calcBin.y × | calc.y | calc.l

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int yylex(void);
extern void yyerror(char *s);
char *binarioStr(float num);
%}
%union {
    float numBin;
}
%token <numBin> BINARIO
%token MAS MENOS PROD DIV OPEN CLOSE ENTER
%type <numBin> expr
%left MAS MENOS
%left PROD DIV
%start input
%%
input:
      /* vacío */
    | input expr ENTER {
        char *b = binarioStr($2);
        printf("Resultado: %s\n\n", b);
        free(b);
    }
    | input error ENTER { yyerror("Syntax error"); }
    ;
```

Figura 6: Código de Yacc para Calculadora de Números Binarios-Parte 1

calcBin.y × | calc.y | calc.l

```
expr:
      BINARIO {
        $$ = $1;
        char *b = binarioStr($$);
        printf("Numero: %s\n", b);
        free(b);
    }
    | MENOS expr          { $$ = -$2; }
    | OPEN expr CLOSE     { $$ = $2; }
    | expr PROD expr      { $$ = $1 * $3; }
    | expr DIV expr       {
          if ($3 == 0.0f) {
              yyerror("Division entre cero");
              YYERROR;
          } else {
              $$ = $1 / $3;
          }
      }
    | expr MAS expr       { $$ = $1 + $3; }
    | expr MENOS expr     { $$ = $1 - $3; }
    ;

%%
```

Figura 7: Código de Yacc para Calculadora de Números Binarios-Parte 2

calcBin.y × | calc.y | calc.l

```
char *binarioStr(float num){
    char *result = (char *)malloc(256);
    int conteo = 0;
    if (!result) return NULL;
    memset(result, 0, 256);
    if (num < 0) {
        result[conteo++] = '-';
        num = -num;
    }
    int parte_entera = (int)num;
    float parte_decimal = num - parte_entera;
    if (parte_entera == 0) {
        result[conteo++] = '0';
    } else {
        char temp[128];
        int i = 0;
        while (parte_entera > 0) {
            temp[i++] = (parte_entera % 2) ? '1' : '0';
            parte_entera /= 2;
        }
        for (int j = i - 1; j >= 0; j--) {
            result[conteo++] = temp[j];
        }
    }
    if (parte_decimal > 0) {
        result[conteo++] = '.';
        int limite = 0;
        while (parte_decimal > 0 && limite < 16) {
            parte_decimal *= 2;
            if (parte_decimal >= 1) {
```

Figura 8: Código de Yacc para Calculadora de Números Binarios-Parte 3

calcBin.y × | calc.y | calc.l

```
        }
        for (int j = i - 1; j >= 0; j--) {
            result[conteo++] = temp[j];
        }
    }
    if (parte_decimal > 0) {
        result[conteo++] = '.';
        int limite = 0;
        while (parte_decimal > 0 && limite < 16) {
            parte_decimal *= 2;
            if (parte_decimal >= 1) {
                result[conteo++] = '1';
                parte_decimal -= 1;
            } else {
                result[conteo++] = '0';
            }
            limite++;
        }
    }
    result[conteo] = '\0';
    return result;
}
int main() {
    printf("Ingrese una expresion binaria: ");
    yyparse();
    return 0;
}
void yyerror(char *s) {
    printf("\nError: %s\n", s);
}
```

Figura 9: Código de Yacc para Calculadora de Números Binarios-Parte 4

3.2.4. Código del Analizador Sintáctico

3.2.5. Gramática

La gramática utilizada para la calculadora de números binarios se define como una gramática libre de contexto que permite la evaluación de expresiones aritméticas con números binarios, operadores binarios y paréntesis. La gramática es la siguiente:

$$
\begin{aligned}
\langle input \rangle &\rightarrow \langle input \rangle\ \langle expr \rangle\ \texttt{ENTER} \mid \epsilon \\
\langle expr \rangle &\rightarrow \texttt{BINARIO} \\
&\quad \mid \texttt{MENOS}\ \langle expr \rangle \\
&\quad \mid \texttt{OPEN}\ \langle expr \rangle\ \texttt{CLOSE} \\
&\quad \mid \langle expr \rangle\ \texttt{PROD}\ \langle expr \rangle \\
&\quad \mid \langle expr \rangle\ \texttt{DIV}\ \langle expr \rangle \\
&\quad \mid \langle expr \rangle\ \texttt{MAS}\ \langle expr \rangle \\
&\quad \mid \langle expr \rangle\ \texttt{MENOS}\ \langle expr \rangle
\end{aligned}
$$

3.2.6. Pruebas Realizadas

Para validar el correcto funcionamiento de la calculadora de números binarios, se realizaron diversas pruebas utilizando expresiones simples y compuestas. Estas pruebas incluyen operaciones básicas de suma, resta, multiplicación y división, el uso de paréntesis y la combinación de números binarios enteros y fraccionarios. Los resultados obtenidos coinciden con los valores esperados, lo que confirma la correcta integración entre el análisis lexicográfico y sintáctico, así como la correcta conversión entre representaciones binaria y decimal.

3.3. Calculadora de Números Romanos

En esta sección se describe el desarrollo de una calculadora aritmética capaz de evaluar expresiones compuestas por números romanos, incluyendo los valores enteros del sistema romano clásico. La calculadora permite realizar operaciones aritméticas básicas utilizando la representación en números romanos, haciendo uso de operadores binarios y paréntesis. La implementación se realizó utilizando las herramientas Lex y Yacc, separando claramente las fases de análisis lexicográfico y sintáctico.

3.3.1. Análisis Lexicográfico (Lex)

El analizador lexicográfico fue desarrollado utilizando Lex y tiene como objetivo reconocer los distintos tokens presentes en las expresiones en números romanos. Entre los

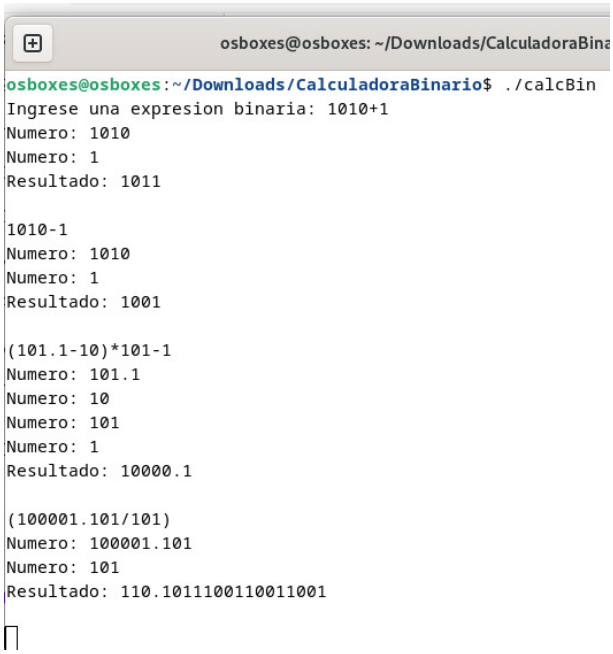

Figura 10: Pruebas realizadas para calculadora de números binarios

tokens identificados se encuentran los números romanos, formados por las letras `I`, `V`, `X`, `L`, `C`, `D` y `M`, así como los operadores aritméticos básicos (`+`, `-`, `*`, `/`), los paréntesis y el carácter de nueva línea que indica el final de una expresión.Para el reconocimiento de números romanos, se definieron expresiones regulares que permiten identificar correctamente valores válidos según la notación romana. Cada número reconocido es convertido internamente a un valor de tipo `int` mediante una función auxiliar, lo que facilita la evaluación de las operaciones aritméticas durante el análisis sintáctico. El analizador lexicográfico también ignora espacios en blanco y caracteres no relevantes, asegurando que únicamente los tokens válidos sean enviados al parser.

3.3.2. Código del Analizador Lexicográfico

3.3.3. Análisis Sintáctico (Yacc)

El análisis sintáctico fue implementado utilizando Yacc, definiendo una gramática libre de contexto que describe la estructura válida de las expresiones aritméticas en números romanos. La gramática permite el uso de operadores binarios de suma, resta, multiplicación y división, así como el uso de paréntesis para modificar la precedencia de las operaciones.

La precedencia de los operadores fue definida de manera que la multiplicación y la división tengan mayor prioridad que la suma y la resta. Aunque la entrada se realiza en formato romano, la evaluación de las expresiones se efectúa utilizando valores enteros. Posteriormente, los resultados son convertidos nuevamente a su representación romana para ser mostrados al usuario. Además, se implementó el manejo de errores semánticos, como la detección de divisiones entre cero, garantizando una ejecución segura del programa.

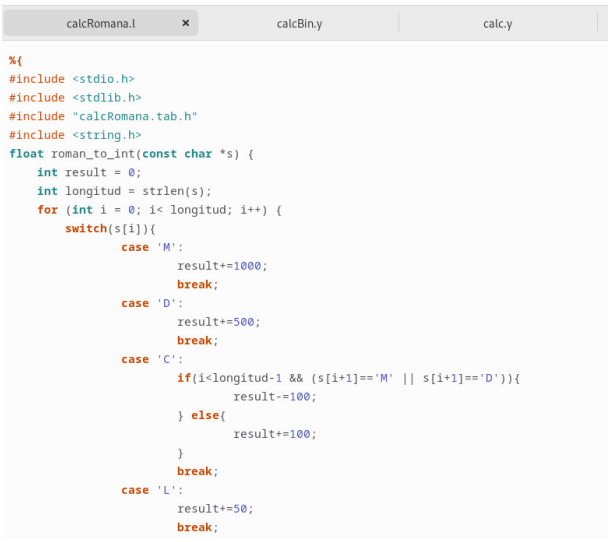

Figura 11: Código de Lex para Calculadora de Números Romanos-Parte 1

Figura 12: Código de Lex para Calculadora de Números Romanos-Parte 2

Figura 13: Código de Lex para Calculadora de Números Romanos-Parte 3

3.3.4. Código del Analizador Sintáctico

calcRomana.y × | calcRomana.l | calcBin.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int yylex(void);
extern void yyerror(char *s);
char *romanoStr(int num);
%}
%union {
    int numRomano;
}
%token <numRomano> ROMANO
%token MAS MENOS PROD DIV OPEN CLOSE ENTER
%type <numRomano> expr
%left MAS MENOS
%left PROD DIV
%start input
%%
input:
      /* vacío */
    | input expr ENTER {
        char *b = romanoStr($2);
        printf("Resultado: %s\n\n", b);
        free(b);
    }
    | input error ENTER { yyerror("Syntax error"); }
    ;
```

Figura 14: Código de Yacc para Calculadora de Números Romanos-Parte 1

calcRomana.y × | calcRomana.l | calcBin.y

```
expr:
    ROMANO {
        $$ = $1;
        char *b = romanoStr($$);
        printf("Numero: %s\n", b);
        free(b);
    }
    | OPEN expr CLOSE     { $$ = $2; }
    | expr PROD expr      { $$ = $1 * $3; }
    | expr DIV expr       {
          if ($3 == 0.0f) {
              yyerror("Division entre cero");
              YYERROR;
          } else {
              $$ = $1 / $3;
          }
      }
    | expr MAS expr       { $$ = $1 + $3; }
    | expr MENOS expr     { $$ = $1 - $3; }
    ;
%%
char *romanoStr(int num){
    char *result = (char *)malloc(10);
    int conteo = 0;
    if (!result) return NULL;
    memset(result, 0, 10);
    int parte_extraida=num%10;
```

Figura 15: Código de Yacc para Calculadora de Números Romanos-Parte 2

3.3.5. Gramática

La gramática utilizada para la calculadora de números romanos se define como una gramática libre de contexto que permite la evaluación de expresiones aritméticas con

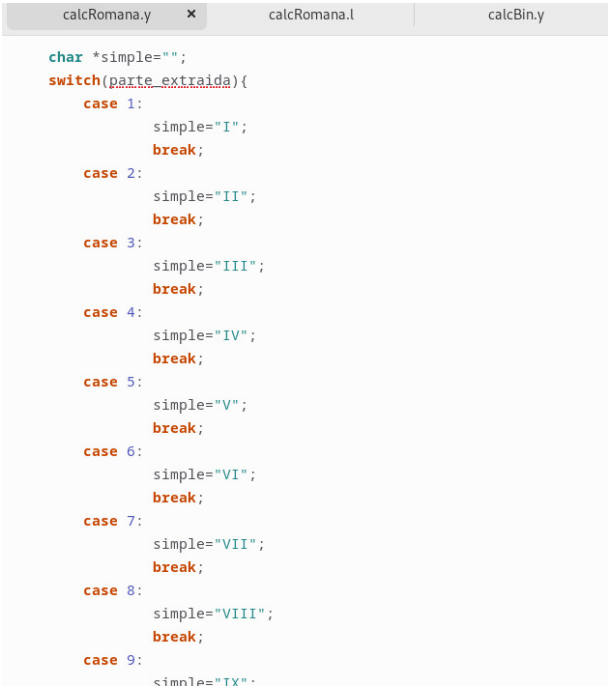

Figura 16: Código de Yacc para Calculadora de Números Romanos-Parte 3

Figura 17: Código de Yacc para Calculadora de Números Romanos-Parte 4

Figura 18: Código de Yacc para Calculadora de Números Romanos-Parte 5

Figura 19: Código de Yacc para Calculadora de Números Romanos-Parte 6

números romanos, operadores binarios y paréntesis. La gramática es la siguiente:

$$
\begin{aligned}
\langle input \rangle &\rightarrow \langle input \rangle \ \langle expr \rangle \ \texttt{ENTER} \mid \epsilon \\
\langle expr \rangle &\rightarrow \texttt{ROMANO} \\
&\mid \texttt{OPEN} \ \langle expr \rangle \ \texttt{CLOSE} \\
&\mid \langle expr \rangle \ \texttt{PROD} \ \langle expr \rangle \\
&\mid \langle expr \rangle \ \texttt{DIV} \ \langle expr \rangle \\
&\mid \langle expr \rangle \ \texttt{MAS} \ \langle expr \rangle \\
&\mid \langle expr \rangle \ \texttt{MENOS} \ \langle expr \rangle
\end{aligned}
$$

3.3.6. Pruebas Realizadas

Para validar el correcto funcionamiento de la calculadora de números romanos, se realizaron diversas pruebas utilizando expresiones simples y compuestas. Estas pruebas incluyen operaciones básicas de suma, resta, multiplicación y división, el uso de paréntesis y la combinación de números romanos de distinto valor. Los resultados obtenidos coinciden con los valores esperados, lo que confirma la correcta integración entre el análisis lexicográfico y sintáctico, así como la correcta conversión entre representaciones romana y decimal.

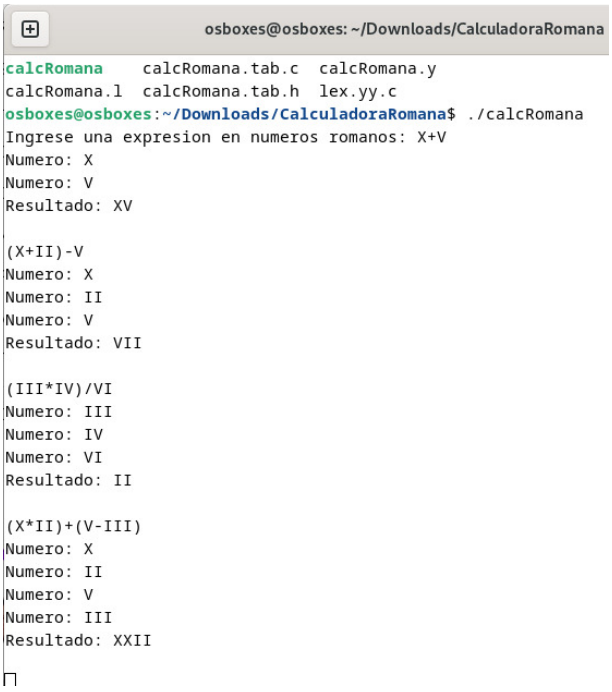

Figura 20: Pruebas realizadas para calculadora de números romanos

3.4. Analizador Lexicográfico y Sintáctico para Calculadora de Números Romanos en Rust

En esta sección se presenta el desarrollo de un analizador lexicográfico y sintáctico para una calculadora de números romanos, pero utilizando el lenguaje de programación Rust (Rust Compiler Team, 2020). La calculadora permitiría realizar operaciones básicas de suma, resta, multiplicación y división, así como el uso de paréntesis para controlar la precedencia de las operaciones.A diferencia de las implementaciones anteriores, en este caso tanto el análisis lexicográfico como el análisis sintáctico fueron desarrollados manualmente dentro del mismo programa, sin el uso de herramientas generadoras externas.

3.4.1. Análisis Lexicográfico

El análisis lexicográfico se encarga de recorrer la cadena de entrada carácter por carácter y convertirla en una secuencia de tokens. Cada token representa un elemento válido de la expresión aritmética, como números romanos, operadores o paréntesis.Para identificar los números romanos, se definió la función `is_roman_char`, la cual valida si un carácter pertenece al conjunto permitido (`I, V, X, L, C, D, M`). Los caracteres romanos consecutivos se agrupan en un único token de tipo `Romano(String)`, permitiendo representar correctamente valores compuestos como `XIV` o `XX`.Además, el analizador reconoce los operadores aritméticos (`+`, `-`, `*`, `/`) y los paréntesis, ignorando espacios en blanco. En caso de encontrar un carácter inválido, el programa genera un error, evitando que expresiones incorrectas sean procesadas en fases posteriores.

3.4.2. Código del Analizador Lexicográfico en Rust

```rust
// Analisis Lexicografico
fn is_roman_char(c: char) -> bool {
    matches!(c, 'I' | 'V' | 'X' | 'L' | 'C' | 'D' | 'M')
}
pub fn lex(input: &str) -> Vec<Token> {
    let mut tokens = Vec::new();
    let mut chars = input.chars().peekable();
    while let Some(&ch) = chars.peek() {
        match ch {
            '+' => { tokens.push(Token::Mas); chars.next(); }
            '-' => { tokens.push(Token::Menos); chars.next(); }
            '*' => { tokens.push(Token::Multiplicar); chars.next(); }
            '/' => { tokens.push(Token::Dividir); chars.next(); }
            '(' => { tokens.push(Token::ParentesisIzquierdo); chars.next(); }
            ')' => { tokens.push(Token::ParentesisDerecho); chars.next(); }
            ' ' | '\t' | '\n' => { chars.next(); }
            _ => {
                if is_roman_char(ch) {
                    let mut roman = String::new();
                    while let Some(&c) = chars.peek() {
                        if is_roman_char(c) {
                            roman.push(c);
                            chars.next();
                        } else {
                            break;
                        }
                    }
                    tokens.push(Token::Romano(roman));
                } else {
                    panic!("Error sintáctico: carácter inválido '{}'", ch);
                }
            }
        }
    }
    tokens
}
```

Figura 21: Código del análisis lexicográfico para números romanos en Rust

3.4.3. Análisis Sintáctico

El análisis sintáctico se encarga de evaluar la expresión aritmética a partir de la lista de tokens generada por el analizador lexicográfico. Para ello, se implementó un parser recursivo descendente que respeta la precedencia de los operadores aritméticos, donde la multiplicación y la división tienen mayor prioridad que la suma y la resta.

El proceso de análisis se divide en varias funciones, cada una encargada de manejar un nivel específico de la gramática:

- `parse_expr`: función principal que inicia el análisis sintáctico.
- `parse_add_sub`: maneja las operaciones de suma y resta.
- `parse_mul_div`: maneja las operaciones de multiplicación y división.
- `parse_primary`: maneja números romanos y expresiones entre paréntesis.

Durante el análisis, los números romanos son convertidos a valores enteros mediante la función `roman_to_int`, lo que permite realizar los cálculos aritméticos de manera sencilla. Una vez obtenida la evaluación final, el resultado entero es convertido nuevamente a su representación romana utilizando la función `int_to_roman`.

Asimismo, se implementó manejo básico de errores sintácticos, como operadores sin operandos, paréntesis desbalanceados o tokens inesperados, lo que mejora la robustez del programa.

3.4.4. Código del Analizador Sintáctico en Rust

```rust
// Analisis Sintáctico
fn parse_expr(tokens: &[Token]) -> (i32, usize) {
    let (value, pos) = parse_add_sub(tokens, 0);
    if pos != tokens.len() {
        panic!("Error sintáctico: tokens sobrantes en la expresión");
    }
    (value, pos)
}
fn parse_add_sub(tokens: &[Token], pos: usize) -> (i32, usize) {
    let (mut value, mut pos) = parse_mul_div(tokens, pos);

    while pos < tokens.len() {
        match tokens[pos] {
            Token::Mas => {
                if pos + 1 >= tokens.len() {
                    panic!("Error sintáctico: '+' sin operando derecho");
                }
                let (rhs, new_pos) = parse_mul_div(tokens, pos + 1);
                value += rhs;
                pos = new_pos;
            }
            Token::Menos => {
                if pos + 1 >= tokens.len() {
                    panic!("Error sintáctico: '-' sin operando derecho");
                }
                let (rhs, new_pos) = parse_mul_div(tokens, pos + 1);
                value -= rhs;
                pos = new_pos;
            }
            _ => break,
        }
    }

    (value, pos)
}
```

Figura 22: Código del análisis sintáctico y evaluación de expresiones en números romanos en Rust-Parte 1

Figura 23: Código del análisis sintáctico y evaluación de expresiones en números romanos en Rust-Parte 2

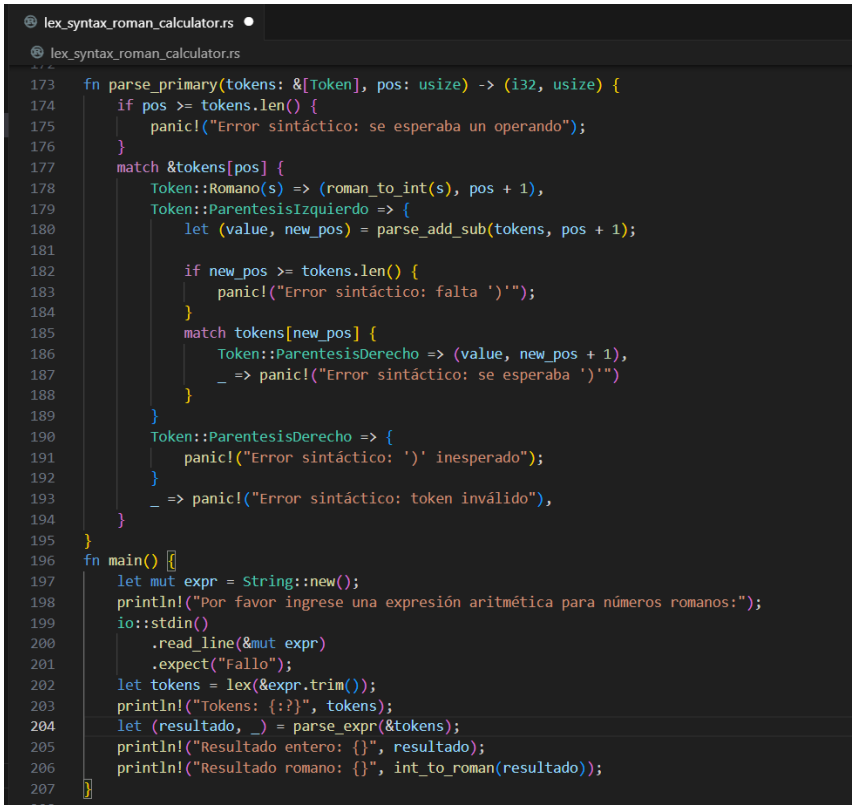

Figura 24: Código del análisis sintáctico y evaluación de expresiones en números romanos en Rust-Parte 3

3.4.5. Gramática

La gramática utilizada para la calculadora en Rust es equivalente a la definida para la versión implementada con Lex y Yacc, y puede expresarse conceptualmente como una gramática libre de contexto:

$$
\begin{aligned}
\langle input \rangle &\rightarrow \langle input \rangle\ \langle expr \rangle\ \texttt{ENTER} \mid \epsilon \\
\langle expr \rangle &\rightarrow \texttt{ROMANO} \\
&\mid \texttt{OPEN}\ \langle expr \rangle\ \texttt{CLOSE} \\
&\mid \langle expr \rangle\ \texttt{PROD}\ \langle expr \rangle \\
&\mid \langle expr \rangle\ \texttt{DIV}\ \langle expr \rangle \\
&\mid \langle expr \rangle\ \texttt{MAS}\ \langle expr \rangle \\
&\mid \langle expr \rangle\ \texttt{MENOS}\ \langle expr \rangle
\end{aligned}
$$

3.4.6. Pruebas Realizadas

Para comprobar el correcto funcionamiento del analizador lexicográfico y sintáctico, se realizaron diversas pruebas utilizando expresiones con números romanos simples y compuestas. Las pruebas incluyen el uso de operadores básicos, expresiones con paréntesis y combinaciones de distintos valores romanos.

Los resultados obtenidos coinciden con los valores esperados, lo que confirma la correcta integración entre el análisis lexicográfico, el análisis sintáctico y los procesos de conversión entre representaciones romana y decimal.

```
PS C:\Users\John\Desktop\AnalizadorLexRomanosRust> .\lex_syntax_roman_calculator.exe
Por favor ingrese una expresión aritmética para números romanos:
V+
Tokens: [Romano("V"), Mas]

thread 'main' (27212) panicked at .\lex_syntax_roman_calculator.rs:137:21:
Error sintáctico: '+' sin operando derecho
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
PS C:\Users\John\Desktop\AnalizadorLexRomanosRust> .\lex_syntax_roman_calculator.exe
Por favor ingrese una expresión aritmética para números romanos:
V-
Tokens: [Romano("V"), Menos]

thread 'main' (4256) panicked at .\lex_syntax_roman_calculator.rs:146:21:
Error sintáctico: '-' sin operando derecho
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
PS C:\Users\John\Desktop\AnalizadorLexRomanosRust> .\lex_syntax_roman_calculator.exe
Por favor ingrese una expresión aritmética para números romanos:
((V-I)/II)+VII
Tokens: [ParentesisIzquierdo, ParentesisIzquierdo, Romano("V"), Menos, Romano("I"), ParentesisDerecho, Dividir, Romano("II"), ParentesisDerecho, Mas, Romano("VII")]
Resultado entero: 9
Resultado romano: IX
PS C:\Users\John\Desktop\AnalizadorLexRomanosRust> .\lex_syntax_roman_calculator.exe
Por favor ingrese una expresión aritmética para números romanos:
((V-I)/II+VII
Tokens: [ParentesisIzquierdo, ParentesisIzquierdo, Romano("V"), Menos, Romano("I"), ParentesisDerecho, Dividir, Romano("II"), Mas, Romano("VII")]

thread 'main' (9236) panicked at .\lex_syntax_roman_calculator.rs:202:17:
Error sintáctico: falta ')'
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
PS C:\Users\John\Desktop\AnalizadorLexRomanosRust>
```

Figura 25: Pruebas realizadas para la calculadora de números romanos implementada en Rust

4. Bibliografía

Referencias

[1] Levine, D. M., Mason, D. N., & Brown, T. A. (2019). *Lex & Yacc* (2nd ed.). O'Reilly Media.

[2] Aho, A. V., & Ullman, J. D. (2007). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[3] Sipser, M. (2012). *Introduction to the Theory of Computation* (3rd ed.). Cengage Learning.

[4] Rust Compiler Team. (2020). *The Rust Programming Language*. Recuperado de https://doc.rust-lang.org/book/