Instituto de Ciencias Básicas e Ingenierías Licenciatura en Ciencias Computacionales

Asignatura:

Autómatas y Compiladores

Catedrático:

Eduardo Cornejo Velázquez

Actividad:

Practica 1

Alumno:

Arana Juárez Raúl

Semestre y grupo:

6o - 3

Período:

Enero – junio 2026

Contenido

COMUNICACIÓN DE REDES
PRACTICA COMPLEMENTARIA

Identificación

Nombre de la Práctica: Análisis de expresiones regulares

No. De Practica: 1 **No. De Sesiones:** 1

No. De integrantes por equipo: 1

Introducción

El análisis de cadenas de símbolos es una parte fundamental en el estudio de los lenguajes formales y la teoría de la computación. A través de una gramática formal es posible definir un lenguaje compuesto por palabras formadas a partir de un alfabeto determinado.

Las expresiones regulares constituyen una herramienta formal que permite describir y reconocer lenguajes regulares de manera precisa y estructurada. En el ámbito de la programación, estas permiten validar, clasificar y analizar cadenas de texto de forma eficiente.

En esta práctica se desarrollaron programas en C++ que permiten analizar cadenas ingresadas por el usuario para clasificarlas como números enteros, palabras en minúsculas, palabras en mayúsculas, identificadores válidos o símbolos, utilizando dos enfoques: estructuras de control tradicionales y expresiones regulares.

Marco Teórico

Una ***Gramática Formal*** definida sobre un alfabeto $\sum$ es una tupla de la forma:

$$\mathbf{G} = \{\sum_{\mathbf{T}}, \sum_{\mathbf{N}}, \mathbf{S}, \mathbf{P}\}$$

donde:

- $\sum_{\mathbf{T}}$ es un alfabeto de símbolos terminales
- $\sum_{\mathbf{N}}$ es un alfabeto de símbolos no terminales
- $\boldsymbol{S}$ es el símbolo inicial de la gramática
- $\mathbf{P}$ es un conjunto de producciones gramaticales

Además, se cumple:

$$\mathbf{S} \in \sum_{\mathbf{N}}$$
$$\sum_{\mathbf{T}} \cap \sum_{\mathbf{N}} = \emptyset$$
$$\sum = \sum_{\mathbf{T}} \cup \sum_{\mathbf{N}}$$

La gramática formal $\mathbf{G}$ permite generan un lenguaje $\mathbf{L} = \{\mathbf{x} \in \sum_{\mathbf{T}}^{*} | \mathbf{S} \overset{*}{\rightarrow} \mathbf{x}\}$
Por lo tanto, las palabras del lenguaje estarán formadas por cadenas de símbolos terminales generadas a partir del símbolo inicial de la gramática utilizando las producciones que la definen.

Una ***expresión regular*** es una notación normalizada para representar lenguajes regulares, es decir, lenguajes generados por gramáticas regulares (Tipo 3). Las expresiones regulares permiten describir con exactitud y sencillez cualquier lenguaje regular.

Lenguaje Utilizado
Expresiones regulares utilizadas
Minusculas
^[a-z]+$
Mayusculas
^[A-Z]+$
Numeros enteros
^[0-9]+$
Identificador valido
^[A-Za-z_][A-Za-z0-9_]*$

Objetivo General

Construir programas en un lenguaje de alto nivel que permitan analizar cadenas de símbolos mediante estructuras de datos y expresiones regulares.

Objetivos Específicos

- Identificar el proceso de análisis de cadenas.

- Aplicar expresiones regulares en C++.
- Clasificar y contabilizar palabras según su tipo.

Desarrollo

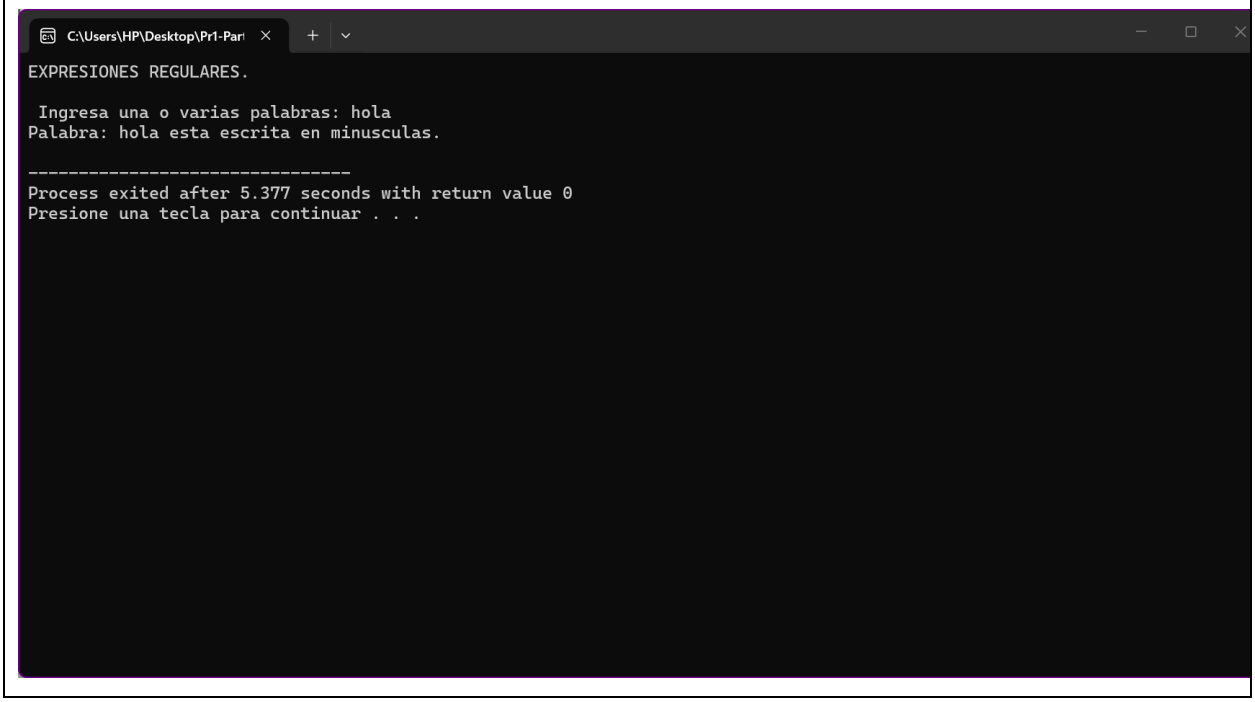
Para realizar esta práctica se utilizó el lenguaje de programación C++, empleando la librería estándar **<regex>** para poder analizar las palabras mediante expresiones regulares. El objetivo fue clasificar las cadenas que el usuario ingresara según ciertas reglas previamente definidas.

El programa comienza solicitando al usuario que escriba una línea de texto que puede contener una o varias palabras separadas por espacios. Para poder capturar toda la línea completa se utilizó la función:

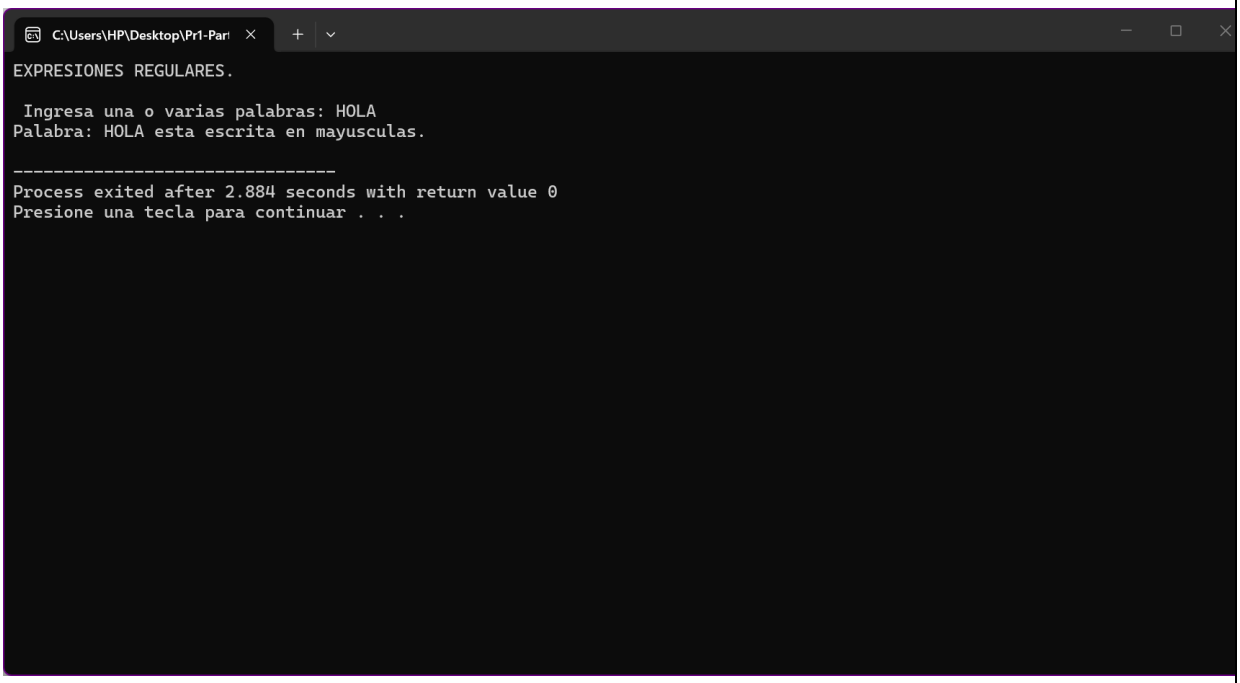
El uso de **getline()** fue necesario porque permite leer toda la entrada, incluyendo espacios, lo que facilita analizar varias palabras en una sola ejecución del programa.

Después de capturar la línea, se definieron las expresiones regulares necesarias para identificar cada tipo de palabra:

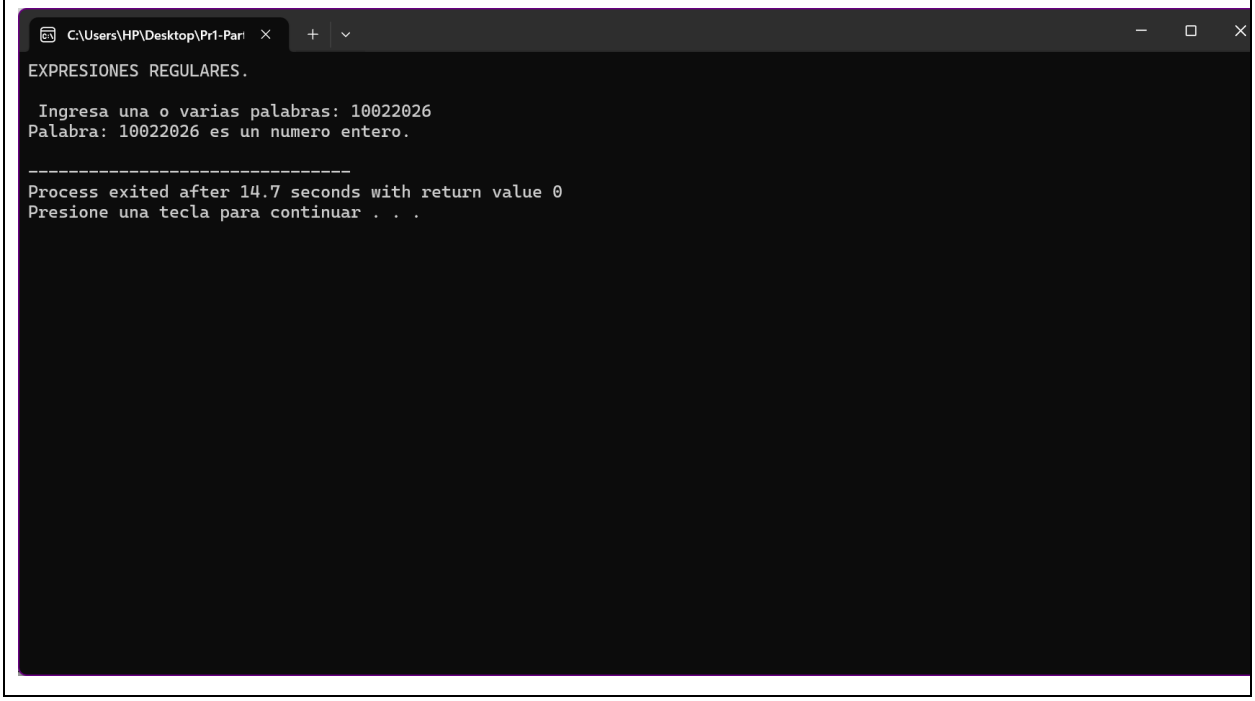
regex minusculas("^[a-z]+$"); : Permite reconocer palabras formadas únicamente por letras minúsculas, desde la primera hasta la última posición.



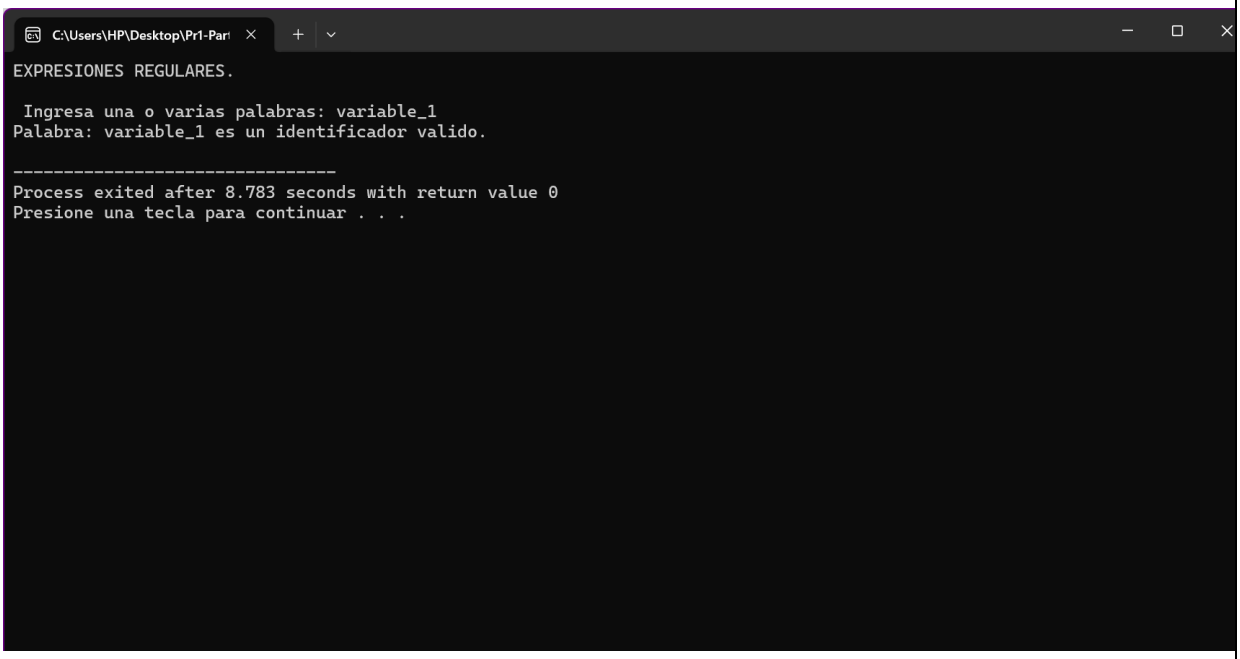
regex mayusculas("^[A-Z]+$"); : Identifica palabras compuestas solamente por letras mayúsculas.



regex numero("^[0-9]+$"); : Detecta números enteros positivos formados únicamente por dígitos del 0 al 9.



regex identificador("^[A-Za-z_][A-Za-z0-9_]*$"); **:** Permite reconocer identificadores válidos, es decir, palabras que comienzan con una letra o guion bajo y que pueden contener letras, números o guion bajo en el resto de la cadena.



Para poder analizar cada palabra por separado se utilizó la clase stringstream, la cual divide la línea en palabras individuales tomando como separador el espacio y finalmente se utilizó un ciclo **while** para recorrer cada palabra extraída

Dentro del ciclo se aplicó la función **regex_match()** para comparar cada palabra con las expresiones regulares definidas. Se utilizó una estructura condicional **if-else if** para determinar a qué categoría pertenece cada palabra y mostrar el resultado correspondiente en pantalla.

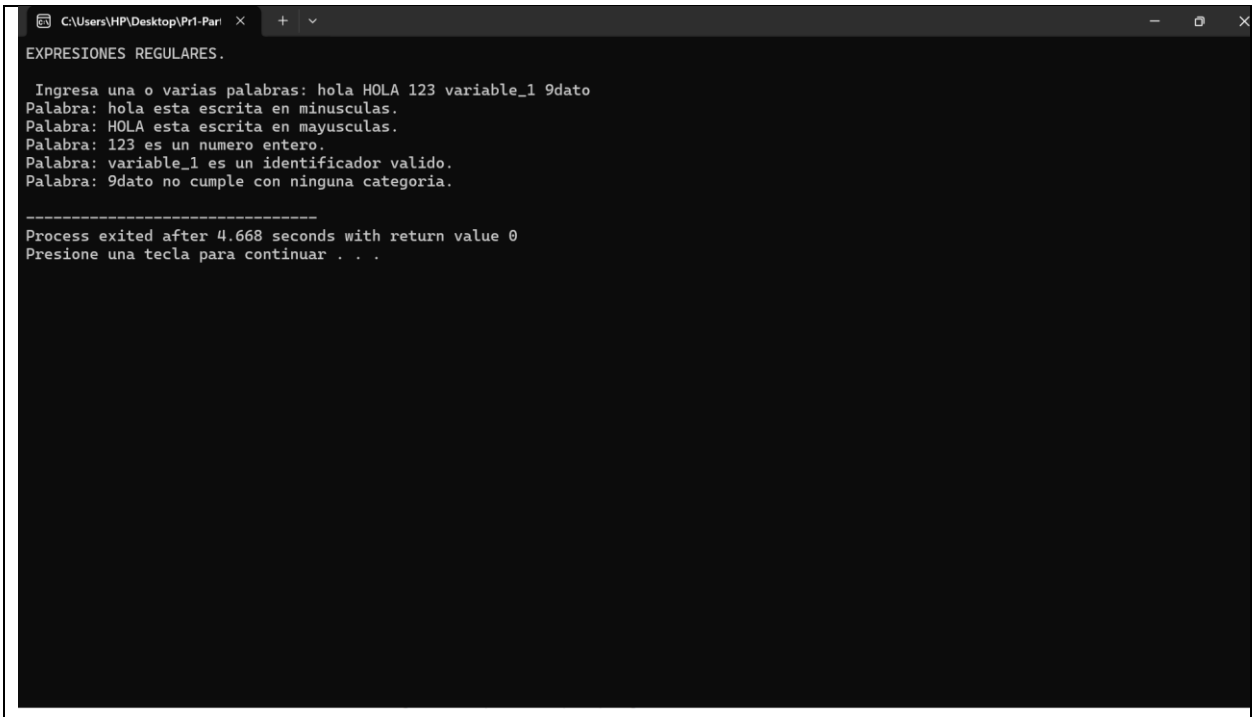
Resultado

El programa funcionó correctamente al analizar tanto una sola palabra como varias palabras escritas en la misma línea. Cada palabra fue evaluada de manera independiente y clasificada según las reglas establecidas en las expresiones regulares.

Por ejemplo, si el usuario ingresa la siguiente línea:

hola HOLA 123 variable_1 9dato

El programa muestra el siguiente resultado:



Se comprobó que el programa reconoce correctamente las palabras formadas solo por letras minúsculas y mayúsculas, así como los números enteros positivos. También identifica adecuadamente los nombres de variables que cumplen con las reglas establecidas para los identificadores. En el caso de cadenas que no cumplen con ningún patrón definido, el programa indica que no pertenecen a ninguna categoría.

El uso de expresiones regulares facilitó considerablemente el proceso de análisis, ya que permitió validar patrones completos de forma directa y organizada, evitando revisar carácter por carácter como en la implementación tradicional.

Enlaces

Parte 1

https://github.com/RaulArana07/Aut-matas-Y-Compiladores/blob/d1d3fc8850c451161104a42239e19f5757dad3a3/practica1/Pr1-Parte1.cpp

Parte 2

https://github.com/RaulArana07/Aut-matas-Y-Compiladores/blob/d1d3fc8850c451161104a42239e19f5757dad3a3/practica1/Pr1-Parte2.cpp

Conclusiones

Se reforzó el conocimiento sobre la importancia de definir correctamente un alfabeto y establecer reglas bien estructuradas para reconocer un lenguaje. Esta práctica ayudó a relacionar la teoría de lenguajes formales con una aplicación práctica en programación, mostrando cómo estos conceptos se utilizan en el análisis léxico de los compiladores y que la implementación mediante expresiones regulares resultó ser una herramienta eficiente, clara y funcional para el análisis de cadenas

Bibliografía

- Giró, J., Vázquez, J., Meloni, B., & Constable, L. (2015). Lenguajes formales y teoría de autómatas. Alfaomega.
- Ruiz Catalán, J. (2010). Compiladores: teoría e implementación. Alfaomega.
- Brookshear, J. G. (1995). Teoría de la Computación: Lenguajes formales, autómatas y complejidad. Addison-Wesley Iberoamericana.