



Universidad Autónoma del Estado de Hidalgo

Instituto de Ciencias Básicas e Ingeniería

Licenciatura en Ciencias Computacionales

Autómatas y Compiladores

Docente: Eduardo Cornejo Velazquez

Práctica 1. Análisis de Expresiones Regulares

Martinez Delgado Baruch's Jesua

Sem: 6 Gpo:3

Fecha de entrega: 10/02/2026



Tabla de contenidos

Contenido

Introduccion.....	2
Marco Teorico	2
Expresiones Regulares	2
Analizador Léxico	3
Aplicaciones de las Expresiones Regulares.....	3
Ventajas de los Lenguajes Regulares	3
Objetivos	4
Desarrollo	4
Resultados	7
Conclusiones.....	8
Bibliografía	8

Introduccion

En el área de los lenguajes de programación y los compiladores, el poder analizar cadenas de símbolos es un proceso común y fundamental en nuestro ámbito para poder identificar secuencias de caracteres que podrían pertenecer a un lenguaje determinado. Este proceso permite reconocer elementos básicos de un lenguaje como números, identificadores o palabras reservadas.

Las expresiones regulares son una herramienta formal que se utiliza para escribir patrones dentro de cadenas de texto. Estas permiten representar lenguajes de manera sencilla y precisa, así facilitando su reconocimiento mediante programas.

Marco Teorico

Expresiones Regulares

Una expresión regular (regex) es una secuencia de símbolos que describe un patrón de texto.

Permite reconocer cadenas pertenecientes a un lenguaje regular.

Operadores básicos

Operador	Nombre	Significado
a	Símbolo	Coincidencia literal
ab	Concatenación	a seguido de b
`a b`	Unión	
a*	Cerradura de Kleene	Cero o más repeticiones
a+	Positiva	Una o más repeticiones
a?	Opcional	Cero o una repetición
()	Agrupación	Prioridad

Ejemplos comunes

Tipo	Expresión regular	Significado
Entero	[0-9]+	Uno o más dígitos
Decimal	[0-9]+\.[0-9]+	Número con punto decimal
Minúsculas	[a-z]+	Letras minúsculas
Mayúsculas	[A-Z]+	Letras mayúsculas
Identificador	[a-zA-Z][a-zA-Z0-9]*	Variable válida
Correo	[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}	Email válido
Símbolo	^[a-zA-Z0-9\s]	Carácter especial

Analizador Léxico

El analizador léxico (scanner) es la primera fase de un compilador. Su función es transformar el código fuente en tokens.

Ejemplo:

```
int edad = 20;
```

Se convierte en:

Lexema	Token
int	Palabra reservada
edad	Identificador
=	Operador
20	Número
;	Delimitador

El analizador léxico utiliza expresiones regulares y autómatas finitos para identificar cada token.

Aplicaciones de las Expresiones Regulares

Las expresiones regulares son ampliamente utilizadas en:

- Compiladores
- Motores de búsqueda
- Validación de formularios
- Bases de datos
- Ciberseguridad
- Procesamiento de lenguaje natural
- Big Data
- Logs de sistemas

Ventajas de los Lenguajes Regulares

- Bajo costo computacional
- Alta velocidad de reconocimiento
- Implementación sencilla
- Base de los compiladores
- Automatización de validaciones

Objetivos

Analizar, diseñar e implementar un sistema de reconocimiento de cadenas basado en lenguajes regulares, mediante el uso de estructuras de datos, algoritmos de procesamiento de texto y expresiones regulares, con la finalidad de identificar, clasificar y validar diferentes tipos de palabras pertenecientes a un lenguaje formal.

Desarrollo

1. Utilizar un lenguaje de programación para construir programas basados en estructuras de datos para analizar cadenas de símbolos con el propósito de identificar:

- Número entero
- Palabras en minúsculas
- Palabras en mayúsculas
- Identificadores (Nombres de variables)

En este primer programa use funciones que verificaban que si las palabras de una cadena de caracteres pertenecían a diferentes categorías como número, palabra minúscula, mayúscula e identificadores. Logres esto al primero separar las cadenas en palabras separadas que pudieran ser analizadas de forma individual.

para separar la cadena use un **istream**, que es una herramienta en C++ para "tokenizar" (dividir) un texto basándose en los espacios en blanco de forma automática.

```
cout << "Introduce el texto a analizar: ";  
if (getline(cin, entrada)) {  
    istringstream iss(entrada);
```

Así mismo cada palabra separada se va a una lista que después entrara a un bucle en el que se someterá a varios condicionales para ver que tipo de palabra es esta.

```
while (iss >> segmento) {  
    lista.push_back(segmento);  
}
```

```
cout << "\n--- Resultados del Analisis ---\n";
    for (const auto& s : lista) {
        cout << s << " -> " << clasificar(s) << endl;
    }
```

Esta será la función a la cual la lista será sometida para saber la naturaleza de las palabras, dando así una clasificación sencilla de las palabras.

```
string clasificar(const string& s) {
    if (s.empty()) return "Vacio";

    bool todosDigitos = true;
    bool todasMinusculas = true;
    bool todasMayusculas = true;
    bool puedeSerIdentificador = true;

    if (isdigit(s[0])) puedeSerIdentificador = false;

    for (char c : s) {
        if (!isdigit(c)) todosDigitos = false;
        if (!islower(c)) todasMinusculas = false;
        if (!isupper(c)) todasMayusculas = false;

        if (!isalnum(c) && c != '_' ) puedeSerIdentificador = false;
    }

    if (todosDigitos) return "Numero entero";
    if (todasMinusculas) return "Palabra en minusculas";
    if (todasMayusculas) return "Palabra en mayusculas";
    if (puedeSerIdentificador) return "Identificador (Variable)";

    return "Simbolos/Mixto";
}
```

2. Definir el alfabeto de símbolos y las expresiones regulares que permita identificar las siguientes tipos de palabras de un lenguaje regular: • Números enteros • Palabras en minúsculas • Palabras en mayúsculas • Identificador (Nombres de variables) • Símbolos

Tipo de Palabra	Expresión Regular (Regex)	Descripción Técnica
Números Enteros	<code>^-?\d+\$</code>	Opcionalmente un signo menos -, seguido de uno o más dígitos.

Tipo de Palabra	Expresión Regular (Regex)	Descripción Técnica
Palabras Minúsculas	<code>^[a-z]+\$</code>	Solo letras de la a a la z, con longitud mínima de uno.
Palabras Mayúsculas	<code>^[A-Z]+\$</code>	Solo letras de la A a la Z, con longitud mínima de uno.
Identificadores	<code>^[a-zA-Z_][a-zA-Z0-9_]*\$</code>	Empieza con letra o _, seguido de letras, números o _.
Símbolos	<code>^[^\w\s]+\$</code>	Detecta caracteres que no sean letras, números ni espacios.

3. Utilizar un lenguaje de programación para construir programas basados en expresiones regulares para analizar cadenas de símbolos con el propósito de clasificar y contabilizar las palabras en las categorías: • Número entero • Palabras en minúsculas • Palabras en mayúsculas • Identificador (Nombre de variable) • Símbolo

Con base a la tabla de expresiones regulares empecé mi programa con ayuda de la función regex que sirve para poner las expresiones regulares deseadas y es tan sencillo como hacer un especie de match entre la palabra y las expresiones especificadas.

```
// Expresiones regulares
regex numero("^-?\d+$");
regex minusculas("^[a-z]+$");
regex mayusculas("^[A-Z]+$");
regex identificador("^[a-zA-Z_][a-zA-Z0-9_]*$");
regex simbolo("^[^\w\s]+$");

while (iss >> palabra) {

    if (regex_match(palabra, numero)) {
        cout << palabra << " -> Numero Entero\n";
        contNumero++;
    }
    else if (regex_match(palabra, minusculas)) {
        cout << palabra << " -> Palabra en Minusculas\n";
        contMinusculas++;
    }
    else if (regex_match(palabra, mayusculas)) {
        cout << palabra << " -> Palabra en Mayusculas\n";
        contMayusculas++;
    }
    else if (regex_match(palabra, identificador)) {
```

```

        cout << palabra << " -> Identificador\n";
        contIdentificador++;
    }
    else if (regex_match(palabra, simbolo)) {
        cout << palabra << " -> Simbolo\n";
        contSimbolo++;
    }
    else {
        cout << palabra << " -> No clasificado\n";
    }
}
}

```

Cada palabra sera clasificada según a que expresion regular sirve, y este programa hara un conteo de las palabras y las ira especificando por el tipo.

Resultados

Ejercicio 1.

```

C:\Users\Baruchs Delgac x + v
Introduce el texto a analizar: Hola a todos mis 54 amigos !

--- Resultados del Analisis ---
Hola -> Identificador (Variable)
a -> Palabra en minusculas
todos -> Palabra en minusculas
mis -> Palabra en minusculas
54 -> Numero entero
amigos -> Palabra en minusculas
! -> Simbolos/Mixto

-----
Process exited after 16.03 seconds with return value 0
Presione una tecla para continuar . . .

```

Ejercicio 3.

```

C:\Users\Baruchs Delgac x + v
Introduce el texto a analizar: HOLA Profesor eduardo le envio 64 saludos !
HOLA -> Palabra en Mayusculas
Profesor -> Identificador
eduardo -> Palabra en Minusculas
le -> Palabra en Minusculas
envio -> Palabra en Minusculas
64 -> Numero Entero
saludos -> Palabra en Minusculas
! -> Simbolo

--- Conteo Final ---
Numeros Enteros: 1
Palabras Minusculas: 4
Palabras Mayusculas: 1
Identificadores: 1
Simbolos: 1

-----
Process exited after 16.93 seconds with return value 0
Presione una tecla para continuar . . . |

```

Conclusiones

Para cerrar tu proyecto, aquí tienes una propuesta de conclusión que resume la transición entre la lógica de programación manual y la teoría de lenguajes regulares, junto con las referencias en formato APA.

Conclusión

El desarrollo de este analizador léxico permite contrastar dos enfoques fundamentales en la informática: la **lógica procedimental** y la **teoría de lenguajes regulares**. Mientras que el uso de funciones nativas como `isdigit` o `islower` en C++ ofrece un control manual y detallado mediante estructuras de control, las **expresiones regulares** proporcionan una sintaxis matemática más compacta y estandarizada para definir patrones complejos.

La implementación demuestra que, independientemente del método, la correcta definición de un **alfabeto** y de las reglas de transición es crucial para evitar ambigüedades, especialmente al diferenciar entre identificadores y palabras reservadas o números. Este ejercicio sienta las bases para entender cómo los compiladores modernos procesan el código fuente antes de ser ejecutado.

Bibliografía

- **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2008).** *Compiladores: Principios, técnicas y herramientas* (2da ed.). Pearson Educación.
- **Joyanes Aguilar, L., & Zahonero Martínez, I. (2008).** *Programación en C++, algoritmos, estructuras de datos y objetos* (2da ed.). McGraw-Hill.
- **Stroustrup, B. (2013).** *The C++ Programming Language* (4th ed.). Addison-Wesley Professional.
- **Regex101. (s.f.).** *Regex101: Build, test, and debug regex*. Recuperado de <https://regex101.com/>