



Instituto Tecnológico de Costa Rica

Escuela de Computación

Compiladores e Intérpretes

Proyecto 1: Análisis Léxico

Estudiantes:

Rafael Alberto Araya Álvarez , José Fabián Zumbado Ruiz

Profesor:

Allan Rodriguez Davila

Verano, 2025

Tabla de Contenidos

Tabla de Contenidos	2
1. Manual de Usuario	3
1.1 Requisitos del Sistema	3
1.2 Instalación	3
1.3 Compilación	4
1.4 Ejecución	4
1.5 Ejemplos de Uso	8
1.6 Detener Docker	11
2. Pruebas de funcionalidad	12
3. Descripción del Problema	12
3.1 Contexto General	12
3.2 Objetivos del Proyecto	12
3.3 Características del Lenguaje	14
Tipos de Datos	12
Palabras Reservadas Especiales	13
Estructuras de Control	13
Operadores	13
Símbolos Especiales	13
3.4 Alcance del Proyecto	14
4. Diseño del Programa	14
5. Librerías Utilizadas	15
5.1 Java Cup Runtime	15
5.2 JFlex	15
5.3 Librerías Estándar de Java	16
5.3.1 java.io	16
5.3.2 java.lang.reflect	16
5.4 Docker	17
5.5 Resumen de Dependencias	17
6. Análisis de Resultados	18
7. Bitácora del Proyecto	19

1. Manual de Usuario

1.1 Requisitos del Sistema

Software Necesario

1. Docker Desktop
 - Versión: 20.10 o superior
 - Sistema operativo: Windows 10/11, macOS, o Linux
2. JFlex
 - Versión: 1.8.2
 - Incluido en el contenedor Docker
3. Java Cup
 - Versión: 0.11b
 - Incluido en el contenedor Docker
4. Java Development Kit (JDK)
 - Versión: 11 o superior
 - Incluido en el contenedor Docker

Hardware Recomendado

- Memoria RAM: 4 GB mínimo, 8 GB recomendado
- Espacio en disco: 2 GB disponibles
- Sistema operativo: Windows 10/11, macOS 10.14+, o Linux (Ubuntu 20.04+)

1.2 Instalación

Paso 1: Clonar o Descargar el Proyecto

```
git clone [URL_DEL_REPOSITORIO]
cd PP1_RafaelAraya_JoseZumbado
```

Paso 2: Verificar Estructura del Proyecto

Asegúrese de que la estructura del proyecto sea la siguiente:

```
PP1_RafaelAraya_JoseZumbado/
├─ programa/...
├─ documentacion/ ...
```

Paso 3: Preparar el Entorno Docker

El proyecto utiliza Docker para garantizar un entorno de compilación consistente.

```
cd programa
docker compose up -d --build
```

La función que cumple este comando es:

- Construir la imagen Docker con todas las dependencias necesarias
- Levantar el contenedor en modo detached (segundo plano)
- Instalar JFlex, Java Cup y JDK

1.3 Compilación

Para compilar manualmente cada componente se tiene que hacer lo siguiente:

```
# Entrar al contenedor
docker compose exec compilador bash

# Generar el lexer
build
```

Verificación de Compilación

Después de compilar, debe ver los siguientes archivos generados:

```
programa/
├── Lexer.java           # Generado por JFlex
├── parser.java          # Generado por Cup
├── sym.java             # Generado por Cup
├── Lexer.class          # Compilado
├── Main.class           # Compilado
├── TestLexer.class      # Compilado
└── ... (otros .class)
```

1.4 Ejecución

Posteriormente a la compilación, con los archivos generados se procede a la ejecución, para esto se debe hacer lo siguiente:

```
# Para ejecutar la prueba
java TestLexer test.txt
```

El archivo test.txt contiene pruebas del código del nuevo lenguaje de programación generado, el cual al momento de ejecutar el comando, éste será analizado por el analizador léxico e imprimirá en consola el resultado.

1.5 Ejemplos de Uso

A continuación mostraremos dos ejemplos de códigos creados en el nuevo lenguaje, ubicados en el test.txt, donde sirven para ejemplificar el funcionamiento del lexer.

1. Código completamente correcto.

```

| Programa que muestra cual numero es mayor.

| Funcion para calcular si una variable entera es mayor a otra.
gift boolean esMayor ¿ int a , int b ?
i
  decide of
    a > b -> i
      return true endl
    !
    a == b -> i
      return false endl
    !
  else -> i
    return false endl
  !
end decide endl
!

| Funcion principal del programa, lee dos variables enteras ejecuta la funcion
de esMayor y valida con un decide of cual variable es mayor o si son iguales.
coal navidad
i
  local int a = get endl
  local int b = get endl

  local boolean flag = esMayor ¿ a , b ? endl

  decide of
    flag == true -> i
      show "A es mayor" endl
    !
  else -> i
    show "B es mayor o igual" endl
    !
  end decide endl

  break endl
!

```

Código:

Salida:

```
root@4d432414c8ea:/app/proyecto# java TestLexer test.txt
```

```
=====
```

```
Probando lexer con archivo: test.txt
```

```
=====
```

NUM	TOKEN	LINEA	COLUMNA	LEXEMA
1	GIFT	3	1	gift
2	BOOLEAN	3	6	boolean
3	ID	3	14	esMayor
4	PARENTHESIS_OPEN	3	22	{
5	INT	3	24	int
6	ID	3	28	a
7	COMMA	3	30	,
8	INT	3	32	int
9	ID	3	36	b
10	PARENTHESIS_CLOSE	3	38	}
11	BLOCK_OPEN	4	1	{
12	DECIDE	5	5	decide
13	OF	5	12	of
14	ID	6	9	a
15	MORE_THAN	6	11	>
16	ID	6	13	b
17	MINUS	6	15	-
18	MORE_THAN	6	16	>
19	BLOCK_OPEN	6	18	{
20	RETURN	7	13	return
21	BOOLEAN_LIT	7	20	true
22	ENDL	7	25	endl
23	BLOCK_CLOSE	8	9	}
24	ID	9	9	a
25	EQUALS	9	11	==
26	ID	9	14	b
27	MINUS	9	16	-
28	MORE_THAN	9	17	>
29	BLOCK_OPEN	9	19	{
30	RETURN	10	13	return
31	BOOLEAN_LIT	10	20	false
32	ENDL	10	26	endl
33	BLOCK_CLOSE	11	9	}
34	ELSE	12	5	else
35	MINUS	12	10	-
36	MORE_THAN	12	11	>
37	BLOCK_OPEN	12	13	{
38	RETURN	13	9	return
39	BOOLEAN_LIT	13	16	false
40	ENDL	13	22	endl
41	BLOCK_CLOSE	14	5	}
42	ID	15	5	end
43	DECIDE	15	9	decide
44	ENDL	15	16	endl
45	BLOCK_CLOSE	16	1	}

46	COAL_TYPE	18	1	coal
47	ID	18	6	navidad
48	BLOCK_OPEN	19	1	i
49	LOCAL	20	5	local
50	INT	20	11	int
51	ID	20	15	a
52	ASSIGN	20	17	=
53	GET	20	19	get
54	ENDL	20	23	endl
55	LOCAL	21	5	local
56	INT	21	11	int
57	ID	21	15	b
58	ASSIGN	21	17	=
59	GET	21	19	get
60	ENDL	21	23	endl
61	LOCAL	23	5	local
62	BOOLEAN	23	11	boolean
63	ID	23	19	flag
64	ASSIGN	23	24	=
65	ID	23	26	esMayor
66	PARENTHESIS_OPEN	23	34	{
67	ID	23	36	a
68	COMMA	23	38	,
69	ID	23	40	b
70	PARENTHESIS_CLOSE	23	42	}
71	ENDL	23	44	endl
72	DECIDE	25	5	decide
73	OF	25	12	of
74	ID	26	9	flag
75	EQUALS	26	14	==
76	BOOLEAN_LIT	26	17	true
77	MINUS	26	22	-
78	MORE_THAN	26	23	>
79	BLOCK_OPEN	26	25	i
80	SHOW	27	13	show
81	STRING_LIT	27	18	"A es mayor"
82	ENDL	27	31	endl
83	BLOCK_CLOSE	28	9	!
84	ELSE	29	5	else
85	MINUS	29	10	-
86	MORE_THAN	29	11	>
87	BLOCK_OPEN	29	13	i
88	SHOW	30	13	show
89	STRING_LIT	30	18	"B es mayor o igual"
90	ENDL	30	39	endl
91	BLOCK_CLOSE	31	9	!
92	ID	32	5	end
93	DECIDE	32	9	decide
94	ENDL	32	16	endl
95	BREAK	34	5	break
96	ENDL	34	11	endl

```

97      BLOCK_CLOSE          35          1          !
-----
✓ Total de tokens: 97
✓ Análisis léxico completado exitosamente

```

En este caso, se puede ver que el análisis léxico se realizó con éxito completamente, validando cada uno de los tokens que en total son 97.

2. Código con errores sintácticos.

Ahora, en este caso, tenemos el mismo código, únicamente incluyendo un par de paréntesis en lugar de los signos de interrogación, algo que no está definido en el lexer.

Código:

```

| Programa que muestra cual numero es mayor.

gift boolean esMayor { int a , int b ?
i
  decide of
    a > b -> i
      return true endl
    !
    a == b -> i
      return false endl
    !
  else -> i
    return false endl
  !
end decide endl
!

coal navidad
i
  local int a = get endl
  local int b = get endl

  local boolean flag = esMayor ( a , b ) endl | Aqui esta el error

  decide of
    flag == true -> i
      show "A es mayor" endl
    !
  else -> i
    show "B es mayor o igual" endl
    !
  end decide endl

```

```
break endl
!
```

Salida:

```
root@45e5fd00aa2c:/app/proyecto# java TestLexer test.txt
```

```
=====
```

```
Probando lexer con archivo: test.txt
```

```
=====
```

NUM	TOKEN	LINEA	COLUMNA	LEXEMA
1	GIFT	3	1	gift
2	BOOLEAN	3	6	boolean
3	ID	3	14	esMayor
4	PARENTHESIS_OPEN	3	22	{
5	INT	3	24	int
6	ID	3	28	a
7	COMMA	3	30	,
8	INT	3	32	int
9	ID	3	36	b
10	PARENTHESIS_CLOSE	3	38	}
11	BLOCK_OPEN	4	1	{
12	DECIDE	5	5	decide
13	OF	5	12	of
14	ID	6	9	a
15	MORE_THAN	6	11	>
16	ID	6	13	b
17	MINUS	6	15	-
18	MORE_THAN	6	16	>
19	BLOCK_OPEN	6	18	{
20	RETURN	7	13	return
21	BOOLEAN_LIT	7	20	true
22	ENDL	7	25	endl
23	BLOCK_CLOSE	8	9	}
24	ID	9	9	a
25	EQUALS	9	11	==
26	ID	9	14	b
27	MINUS	9	16	-
28	MORE_THAN	9	17	>
29	BLOCK_OPEN	9	19	{
30	RETURN	10	13	return
31	BOOLEAN_LIT	10	20	false
32	ENDL	10	26	endl
33	BLOCK_CLOSE	11	9	}
34	ELSE	12	5	else

35	MINUS	12	10	-
36	MORE_THAN	12	11	>
37	BLOCK_OPEN	12	13	i
38	RETURN	13	9	return
39	BOOLEAN_LIT	13	16	false
40	ENDL	13	22	endl
41	BLOCK_CLOSE	14	5	!
42	ID	15	5	end
43	DECIDE	15	9	decide
44	ENDL	15	16	endl
45	BLOCK_CLOSE	16	1	!
46	COAL_TYPE	18	1	coal
47	ID	18	6	navidad
48	BLOCK_OPEN	19	1	i
49	LOCAL	20	5	local
50	INT	20	11	int
51	ID	20	15	a
52	ASSIGN	20	17	=
53	GET	20	19	get
54	ENDL	20	23	endl
55	LOCAL	21	5	local
56	INT	21	11	int
57	ID	21	15	b
58	ASSIGN	21	17	=
59	GET	21	19	get
60	ENDL	21	23	endl
61	LOCAL	23	5	local
62	BOOLEAN	23	11	boolean
63	ID	23	19	flag
64	ASSIGN	23	24	=
65	ID	23	26	esMayor
Caracter no reconocido en línea 23: (
66	ID	23	36	a
67	COMMA	23	38	,
68	ID	23	40	b
Caracter no reconocido en línea 23:)				
69	ENDL	23	44	endl
70	DECIDE	25	5	decide
71	OF	25	12	of
72	ID	26	9	flag
73	EQUALS	26	14	==
74	BOOLEAN_LIT	26	17	true
75	MINUS	26	22	-
76	MORE_THAN	26	23	>
77	BLOCK_OPEN	26	25	i
78	SHOW	27	13	show
79	STRING_LIT	27	18	"A es mayor"
80	ENDL	27	31	endl
81	BLOCK_CLOSE	28	9	!
82	ELSE	29	5	else
83	MINUS	29	10	-

```

84     MORE_THAN      29      11      >
85     BLOCK_OPEN     29      13      i
86     SHOW           30      13      show
87     STRING_LIT     30      18      "B es mayor o igual"
88     ENDL           30      39      endl
89     BLOCK_CLOSE    31      9       !
90     ID             32      5       end
91     DECIDE         32      9       decide
92     ENDL           32      16      endl
93     BREAK          34      5       break
94     ENDL           34      11      endl
95     BLOCK_CLOSE    35      1       !

```

✓ Total de tokens: 95

✓ Análisis léxico completado exitosamente

Al ejecutar el análisis léxico, se puede ver que en la línea 63 hay dos caracteres no reconocidos, pero no se detiene la ejecución, por que el manejo de errores actuales al encontrar un error, salta a la siguiente línea.

1.6 Detener Docker

```

# Detener el contenedor
docker compose down

# Detener y eliminar volúmenes
docker compose down -v

```

2. Pruebas de funcionalidad

El video viene junto a la carpeta adjunta para la entrega, en la carpeta “/documentacion”.

3. Descripción del Problema

3.1 Contexto General

El presente proyecto implementa la fase de Análisis Léxico para un lenguaje de programación imperativo. Este lenguaje tiene el objetivo de permitir realizar operaciones básicas para la configuración de un chip.

3.2 Objetivos del Proyecto

El objetivo principal es desarrollar un scanner (analizador léxico) utilizando la herramienta JFlex que sea capaz de:

1. Identificar lexemas: Reconocer todos los elementos léxicos definidos en la gramática del lenguaje.
2. Generar símbolos: Retornar tokens de tipo Symbol utilizando la directiva %cup y la clase sym.
3. Reportar errores: Detectar y reportar errores léxicos de manera informativa.
4. Recuperación de errores: Implementar la técnica de Recuperación en Modo Pánico (reportar el error en la línea actual y continuar con la siguiente).
5. Generar salida: Escribir en un archivo todos los tokens encontrados con su información asociada.

3.3 Características del Lenguaje

El lenguaje imperativo en desarrollo incluye las siguientes características:

Tipos de Datos

- Enteros (int): Números enteros
- Flotantes (float): Números de punto flotante

- Booleanos (boolean): Valores true o false
- Caracteres (char): Caracteres individuales entre comillas simples
- Cadenas (string): Secuencias de caracteres entre comillas dobles
- Arreglos bidimensionales: Arrays estáticos de tipo char e int

Palabras Reservadas Especiales

- navidad: Función principal main
- gift: Declaración de funciones
- world: Declaración de variables globales
- local: Declaración de variables locales
- coal: Tipo de retorno void (carbón para los niños traviesos)
- show: Función de salida (escribir en consola)
- get: Función de entrada (leer desde consola)

Estructuras de Control

- decide of _ end decide: Estructura condicional tipo switch
- loop _ exit when _ end loop: Bucle tipo Ada
- for: Bucle tipo C/Java
- return: Retorno de funciones
- break: Salida de ciclos

Operadores

Aritméticos:

- Binarios: +, -, *, /, // (división entera), % (módulo), ^ (potencia)
- Unarios: - (negativo), ++ (incremento), -- (decremento)

Relacionales:

- <, <=, >, >=, ==, !=

Lógicos:

- @ (AND/conjunción)
- ~ (OR/disyunción)
- Σ (NOT/negación)

Símbolos Especiales

- Bloques de código: ¡ (apertura) y ! (cierre) - reemplazan las llaves tradicionales
- Paréntesis: ¿ (apertura) y ? (cierre) - reemplazan los paréntesis tradicionales
- Fin de sentencia: endl - reemplazan el punto y coma tradicional

- Comentarios:
 - Una línea: | comentario
 - Múltiples líneas: € comentario multilínea ³

3.4 Alcance del Proyecto

El analizador léxico debe ser capaz de:

- Leer un archivo fuente escrito en el lenguaje
- Tokenizar el contenido identificando cada elemento léxico
- Generar tokens con la siguiente información:
 - Tipo de token: Identificador numérico del terminal
 - Lexema: Texto original del token
 - Línea: Número de línea donde aparece
 - Columna: Número de columna donde comienza
- Reportar errores léxicos indicando:
 - Línea del error
 - Carácter o secuencia no reconocida
- Continuar el análisis después de encontrar errores (modo pánico)

4. Diseño del Programa

El diseño del programa está basado en la implementación de un análisis léxico usando la herramienta de JFlex, integrado con %cup. El objetivo principal de este diseño es realizar un análisis de los lexemas definidos en la gramática que hemos creado, la cual satisface lo solicitado en el enunciado del proyecto además de generar los tokens con la información completa permitiendo la detección de errores.

El programa fue diseñado modularmente, separando el analizador léxico, la definición de los tokens y el programa de test. El analizador léxico se encarga únicamente de la lectura del archivo de prueba y para la generación de los tokens.

Una de las decisiones más importantes del diseño es que únicamente se reconozcan los símbolos del lenguaje, en lugar de otros, cualquier carácter que no pertenezca a la especificación es tratado como un error léxico, esto garantiza que se cumpla la estructura establecida en la gramática.

Para manejar los errores léxicos, se implementa la recuperación de modo pánico, el cual consiste en reportar el error en la línea donde ocurre y continúa el análisis sin detener la ejecución. Esto permite ver múltiples errores en una pasada del archivo fuente.

Las expresiones regulares definidas en el lexer las utiliza JFlex para construir internamente autómatas finitos no deterministas, los cuales se transforman en finitos deterministas

optimizados. Con estos autómatas se genera el `Lexer.java`, que se encarga de ejecutar el análisis léxico sobre el archivo de prueba.

Cada vez que se reconoce un lexema correcto, el lexer genera un objeto `Symbol`, que utiliza los identificadores definidos en la clase `sym` e incluyen información sobre el token, lexema, línea y columna donde aparece.

Otro aspecto importante es la implementación de contenedores mediante docker para ejecutar el programa, esto facilita el uso del analizador léxico en cualquier entorno, sin la necesidad de instalar Java, JDK, JFlex y Cup en la máquina principal del usuario.

Este diseño permite que el analizador léxico sea escalable para integrarse en fases posteriores del compilador.

5. Librerías Utilizadas

5.1 Java Cup Runtime

Nombre: Java Cup Runtime Library Versión: 0.11b

Propósito: Proporcionar las clases necesarias para que el lexer retorne objetos `Symbol`

Clases utilizadas:

- `java_cup.runtime.Symbol`: Clase base para tokens
 - Constructor: `Symbol(int id, int left, int right, Object value)`
 - `id`: Tipo de token (desde la clase `sym`)
 - `left`: Línea del token
 - `right`: Columna del token
 - `value`: Lexema del token

Uso en el proyecto:

```
import java_cup.runtime.*;

// En lexer.flex
return new Symbol(sym.INT, yyline+1, yycolumn+1, yytext());
```

Instalación: Incluida en el contenedor Docker, disponible en el classpath.

5.2 JFlex

Nombre: JFlex - The Fast Scanner Generator for Java Versión: 1.8.2

Propósito: Generar el analizador léxico a partir de especificaciones de expresiones regulares

Características utilizadas:

- Generación de DFA optimizado
- Soporte Unicode completo
- Seguimiento de línea y columna
- Integración con Cup mediante %cup

Comando de uso:

```
jflex lexer.flex
```

Entrada: lexer.flex (especificación) Salida: Lexer.java (código generado)

5.3 Librerías Estándar de Java

5.3.1 java.io

Propósito: Manejo de entrada/salida de archivos y flujos

Clases utilizadas:

FileReader: Lectura de archivos de texto

```
FileReader fileReader = new FileReader("test.txt");  
Lexer lexer = new Lexer(fileReader);
```

InputStreamReader: Lectura desde entrada estándar

```
Lexer lexer = new Lexer(new InputStreamReader(System.in));
```

IOException: Manejo de excepciones de I/O

```
catch (IOException e) {  
    System.err.println("Error de lectura: " + e.getMessage());  
}
```

FileNotFoundException: Manejo de archivos no encontrados

```
catch (FileNotFoundException e) {  
    System.err.println("Archivo no encontrado: " + e.getMessage());  
}
```

5.3.2 java.lang.reflect

Propósito: Reflexión para obtener nombres de constantes de la clase sym.

Uso en TestLexer.java:

```
private static String getTokenName(int symCode) {
    java.lang.reflect.Field[] fields = sym.class.getDeclaredFields();
    for (java.lang.reflect.Field field : fields) {
        if (field.getType() == int.class) {
            int value = field.getInt(null);
            if (value == symCode) {
                return field.getName(); // Retorna "INT", "FLOAT", etc.
            }
        }
    }
    return "SYM_" + symCode;
}
```

Permite mostrar nombres legibles de tokens ("INT") en lugar de códigos numéricos (4).

5.4 Docker

Propósito: Crear un entorno de desarrollo reproducible y consistente

Componentes:

1. Dockerfile:
 - Imagen base: OpenJDK 11 para correr Java
 - Instalación de JFlex
 - Instalación de Java Cup
 - Configuración del classpath
2. docker-compose.yml:
 - Definición del servicio "compilador"
 - Montaje de volúmenes para acceso a archivos del proyecto
 - Configuración de red y puertos

Ventajas y Justificación:

- Portabilidad: Funciona igual en Windows, macOS y Linux
- Aislamiento: No interfiere con el sistema host
- Reproducibilidad: Todos los desarrolladores tienen el mismo entorno
- Simplicidad: No requiere instalación manual de dependencias

5.5 Resumen de Dependencias

Librería	Versión	Propósito	Instalación
JFlex	1.8.2	Generador de lexer	Docker
Java Cup	0.11b	Runtime de parser	Docker

JDK	11+	Compilación y ejecución	Docker
Docker	20.10+	Contenedorización	Manual

6. Análisis de Resultados

El sistema actualmente desarrollado del analizador léxico cumple con los siguientes requisitos:

a) El sistema debe leer un archivo fuente.

El programa mediante el comando de ejecución posteriormente a la compilación, específicamente en los puntos 1.4 Ejecución y en el punto 1.5 Ejemplos de uso de este uso, se muestra que el programa se ejecuta y lee el archivo de código fuente test.txt.

b) Se debe escribir en un archivo todos los tokens encontrados, identificador asociado con el lexema.

El programa al momento de ejecutarse como se muestra en el punto 1.4 de este documento, no solo imprime en consola el token, fila, columna y lexema que encuentra dentro del test.txt, sino que también genera un archivo de salida llamado tokens_output.txt, donde se encuentra la misma información.

c) Reportar y manejar los errores léxicos encontrados. Debe utilizar la técnica de Recuperación en Modo Pánico (error en línea y continúa con la siguiente).

El programa, al momento de encontrar un error léxico indica, mediante un mensaje de error, que hay un carácter no válido, lo que cumple con la técnica del Modo Pánico.

Con estos resultados, se puede ver que se ha cumplido con lo solicitado.

7. Bitácora del Proyecto

La bitácora va desde el último commit al primero:

```
commit 978bc32695fb68fb9fa42e029a183ce895b2b074 (HEAD -> main,  
origin/main, origin/HEAD)
```

Author: Rafael Araya

<143631166+RafaAraya14@users.noreply.github.com>

Date: Sun Dec 21 16:39:27 2025 -0600

Comentarios.

```
commit 75a77430e956456c48e08784673df4ed88d2c888 (HEAD -> main,  
origin/main, origin/HEAD)
```

Author: Rafael Araya

<143631166+RafaAraya14@users.noreply.github.com>

Date: Sun Dec 21 15:33:14 2025 -0600

Cambio en el test para la docu

```
commit 490c7ad43843d37c96cf151a3505caac7ce43d1d
```

Author: Joche3023 <j.zumbado.1@estudiantec.cr>

Date: Sat Dec 20 18:54:58 2025 -0600

Add la escritura de los tokens a un archivo tokens_output.txt

```
commit 950a38e3a7856eef7cb6cd35afdaa5ee38f8423e
```

Author: Joche3023 <j.zumbado.1@estudiantec.cr>

Date: Sat Dec 20 18:36:30 2025 -0600

Add lexemas faltantes y llenar el info.txt

```
commit bccd731d4c594e2c9080558a74894e991a4bbb10
```

Author: Rafael Araya

<143631166+RafaAraya14@users.noreply.github.com>

Date: Fri Dec 19 15:10:38 2025 -0600

Parte del Lexer Rafael

```
commit 75ae8ae9af1f0c787036a0279a85ef4946df8a56
Author: Joche3023 <j.zumbado.1@estudiantec.cr>
Date:   Fri Dec 19 13:39:20 2025 -0600
```

Add lexemas al lexer y cambiar el README

```
commit fd758fda9b85efd42301ff1cef2a3971d004a6ce
Author: Joche3023 <j.zumbado.1@estudiantec.cr>
Date:   Thu Dec 18 18:03:55 2025 -0600
```

Add .gitignore para archivos generados y temporales

```
commit 4c0e93c939df6295bb8dbd9571c0eb7b243827cf
Author: Joche3023 <j.zumbado.1@estudiantec.cr>
Date:   Thu Dec 18 16:50:15 2025 -0600
```

Primer commit