



República Bolivariana de Venezuela
Universidad Nacional Experimental de Guayana
Vicerrectorado Académico
Coordinación General de Pregrado

Tema 4

Docente:

Ing. Felix Marquez

Lenguajes y Compiladores

sección 01

Autores:

José García V-28.385.691

Héctor Tovar V-30.335.783

Jenfer Martínez V-30.577.104

Ingeniería informática

Puerto Ordaz, febrero de 2025

1. Defina y presente ejemplos de autómatas de pilas.

Autómata de pila (PDA)

Un autómata de pila (PDA, por sus siglas en inglés) es un modelo computacional que extiende la capacidad de los autómatas finitos (AFD y AFND) al incorporar una pila (stack) como memoria auxiliar. Esta pila permite a la PDA almacenar y manipular datos de manera LIFO (Last In, First Out), lo que le otorga la capacidad de reconocer lenguajes más complejos que los que pueden reconocer los autómatas finitos.

Hopcroft y Ullman (1979) definen formalmente un PDA como una tupla de 7 elementos:

$$(Q, \Sigma, \Gamma, \delta, q_0, Z, F)$$

Donde:

Q: Conjunto finito de estados.

Σ : Alfabeto de entrada (símbolos que la PDA puede leer).

Γ : Alfabeto de la pila (símbolos que se pueden almacenar en la pila).

δ : Función de transición, que define cómo la PDA se mueve de un estado a otro y cómo modifica la pila en función del estado actual, el símbolo de entrada y el símbolo en la cima de la pila.

q_0 : Estado inicial.

Z: Símbolo inicial de la pila.

F: Conjunto de estados finales.

En esencia, un PDA opera leyendo símbolos de entrada y, basándose en su estado actual y el símbolo en la cima de la pila, decide su siguiente estado y la operación a realizar en la pila (apilar, desapilar o mantener el mismo símbolo). Esta capacidad de manipulación de la pila es lo que le permite a la PDA reconocer lenguajes más allá del alcance de los autómatas finitos.

Ejemplo:

Se desarrolló un autómata de pila para palabras reservadas en python, el cual consta de:

```

class Automata:
    def __init__(self):
        self.transitions = {

            #(estado_actual, símbolo_leído, símbolo_en_la_pila): (nuevo_estado, símbolos_a_apilar)

            ('q0', 'i', 'Z0'): ('q1', ['I', 'Z0']),      # 'i' como inicio de 'if' o 'in'
            ('q1', 'f', 'I'): ('q2', []),                # 'if'
            ('q1', 'n', 'I'): ('q3', []),                # 'in'
            ('q0', 'e', 'Z0'): ('q4', ['E', 'Z0']),      # 'e' como inicio de 'elif' o 'else'
            ('q4', 'l', 'E'): ('q5', ['L', 'E']),        # 'el'
            ('q5', 'i', 'L'): ('q6', ['I2', 'L']),       # 'eli'
            ('q6', 'f', 'I2'): ('q2', []),               # 'elif'
            ('q5', 's', 'L'): ('q7', ['S', 'L']),        # 'els'
            ('q7', 'e', 'S'): ('q2', []),               # 'else'
            ('q0', 'w', 'Z0'): ('q8', ['W', 'Z0']),      # 'w' como inicio de 'while'
            ('q8', 'h', 'W'): ('q9', ['H', 'W']),        # 'wh'
            ('q9', 'i', 'H'): ('q10', ['I3', 'H']),      # 'whi'
            ('q10', 'l', 'I3'): ('q11', ['L2', 'I3']),   # 'whil'
            ('q11', 'e', 'L2'): ('q2', []),              # 'while'
            ('q0', 'f', 'Z0'): ('q12', ['F', 'Z0']),    # 'f' como inicio de 'for'
            ('q12', 'o', 'F'): ('q13', ['O', 'F']),      # 'fo'
            ('q13', 'r', 'O'): ('q2', []),              # 'for'
            ('q0', 'r', 'Z0'): ('q14', ['R', 'Z0']),    # 'r' como inicio de 'return'
            ('q14', 'e', 'R'): ('q15', ['E2', 'R']),    # 're'
            ('q15', 't', 'E2'): ('q16', ['T', 'E2']),   # 'ret'
            ('q16', 'u', 'T'): ('q17', ['U', 'T']),     # 'retu'
            ('q17', 'r', 'U'): ('q18', ['R2', 'U']),    # 'retur'
            ('q18', 'n', 'R2'): ('q2', []),             # 'return'
        }
        self.initial_state = 'q0'
        self.accept_states = {'q2'} # q2 es el estado de aceptación
        self.stack = ['Z0'] # Z0 es el símbolo inicial de la pila

```

- self.transitions = { ... }: Diccionario que contiene las transiciones del autómata.
 - Clave: Tupla (estado_actual, símbolo_leído, símbolo_en_la_pila).
 - Valor: Tupla (nuevo_estado, símbolos_a_apilar).
 - Ejemplo: ('q0', 'i', 'Z0'): ('q1', ['I', 'Z0'])
- self.initial_state = 'q0': Estado inicial del autómata.
- self.accept_states = 'q2': Conjunto de estados de aceptación del autómata.
- self.stack = ['Z0']: Inicializa la pila del autómata con el símbolo 'Z0'.

```

○○○

def process_input(self, input_string):
    current_state = self.initial_state
    self.stack = ['Z0']

    for symbol in input_string:
        key = (current_state, symbol, self.stack[-1])
        if key in self.transitions:
            new_state, push_symbols = self.transitions[key]
            current_state = new_state
            self.stack.pop()
            self.stack.extend(reversed(push_symbols))
        else:
            return False # La cadena no es reconocida

    return current_state in self.accept_states # Acepta si está en un estado final

```

Este método recibe una cadena de entrada (`input_string`) y simula la ejecución del autómata sobre dicha cadena.

El estado actual del autómata se inicializa con el estado inicial.

La pila se reinicia al símbolo inicial. Es fundamental resetear la pila para cada nueva cadena que se procesa.

Se itera sobre cada símbolo de la cadena de entrada.

Se genera la clave para buscar la transición en el diccionario `self.transitions`. Esta clave representa la configuración actual del autómata.

Si existe una transición definida para la configuración actual:

- Se obtienen el nuevo estado y los símbolos a apilar.
- Se actualiza el estado actual del autómata.
- Se desapila el símbolo del tope de la pila.
- Se apilan los nuevos símbolos en la pila. Es crucial que los símbolos se apilen en orden inverso al que aparecen en la tupla `push_symbols`.

Si no existe una transición para la configuración actual, la cadena no es reconocida, y la función retorna "False".

Tras procesar toda la cadena, la función verifica si el autómata se encuentra en un estado de aceptación.

Ejemplo de ejecución:

El autómata reconoce palabras como: if, else, elif, for, while, return

```
Ingrese una palabra reservada: in
'in': Rechazada
Ingrese una palabra reservada: or
'or': Rechazada
Ingrese una palabra reservada: &&
'&&': Rechazada
Ingrese una palabra reservada: if
'if': Aceptada
Ingrese una palabra reservada: else
'else': Aceptada
Ingrese una palabra reservada: for
'for': Aceptada
```

Conclusión

Los autómatas de pila representan un avance significativo respecto a los autómatas finitos, ya que su capacidad para manejar memoria dinámica les permite abordar problemas más complejos y estructurados. Su importancia radica en su aplicación en áreas como compiladores, procesamiento de lenguajes naturales y validación de sintaxis, donde la jerarquía y el anidamiento son críticos. Aunque su complejidad operativa es mayor que la de los AFD y AFND, su poder computacional los convierte en una herramienta indispensable en la teoría de la computación y sus aplicaciones prácticas.

2) Construya un lexer para expresiones aritméticas basado en regex. En el Informe debe indicar paso a paso la construcción de su lexer y 3 ejemplos de ejecución.

Un lexer es una herramienta fundamental en el análisis léxico de un lenguaje, encargado de transformar una secuencia de caracteres en una secuencia de tokens. En este informe, se describe la construcción de un lexer para expresiones aritméticas utilizando expresiones regulares (regex).

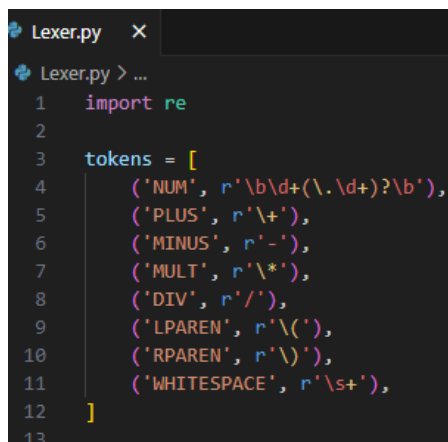
Definición de Tokens

Para reconocer correctamente los componentes de una expresión aritmética, se definen los siguientes tokens:

- **NUM**: Representa números enteros o decimales (ej. 42, 3.14).
- **PLUS**: Representa el operador de suma (+).
- **MINUS**: Representa el operador de resta (-).
- **MULT**: Representa el operador de multiplicación (*).
- **DIV**: Representa el operador de división (/).
- **LPAREN**: Representa el paréntesis izquierdo (()
- **RPAREN**: Representa el paréntesis derecho ())
- **WHITESPACE**: Representa espacios en blanco (que serán ignorados).

Implementación del Lexer en Python

Se utiliza el módulo `re` para definir y aplicar las expresiones regulares

A screenshot of a code editor showing a Python script named 'Lexer.py'. The script defines a list of tokens using regular expressions. The tokens are: NUM (integers or decimals), PLUS (+), MINUS (-), MULT (*), DIV (/), LPAREN (left parenthesis), RPAREN (right parenthesis), and WHITESPACE (spaces).

```
1 import re
2
3 tokens = [
4     ('NUM', r'\b\d+(\.\d+)?\b'),
5     ('PLUS', r'\+'),
6     ('MINUS', r'\-'),
7     ('MULT', r'\*'),
8     ('DIV', r'\/'),
9     ('LPAREN', r'\('),
10    ('RPAREN', r'\)'),
11    ('WHITESPACE', r'\s+'),
12 ]
13
```

Luego se define una función que se encarga de analizar una cadena de entrada (`input_text`) y dividirla en tokens utilizando expresiones regulares.

```
def lexer(input_text):
    token_regex = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in tokens)
    for match in re.finditer(token_regex, input_text):
        kind = match.lastgroup
        value = match.group(kind)
        if kind != 'WHITESPACE':
            yield kind, value
```

Construcción de la expresión regular combinada

```
token_regex = '|'.join(f'(?P<{ name}>{ pattern})' for name, pattern in tokens)
```

Se crea una única expresión regular que combina todas las definiciones de tokens de la lista tokens. Cada token se define con un grupo con nombre (?P<nombre>), lo que permite identificar qué tipo de token ha sido encontrado.

Búsqueda de coincidencias en la entrada

```
for match in re.finditer(token_regex, input_text):
```

La función re.finditer() busca todas las coincidencias en input_text utilizando la expresión regular combinada. Cada vez que encuentra una coincidencia, devuelve un objeto match.

Identificación del token y su valor

```
kind = match.lastgroup
```

```
value = match.group(kind)
```

Las funciones match.lastgroup obtiene el nombre del grupo coincidente (es decir, el tipo de token como NUM, PLUS, etc.) mientras que match.group(kind) obtiene el valor real de la coincidencia en la entrada.

Filtrado de espacios en blanco

```
if kind != 'WHITESPACE':
```

```
yield kind, value
```

Si el token no es un espacio en blanco (WHITESPACE), se devuelve como un par (tipo, valor). Se utiliza yield para devolver los tokens de manera secuencial sin almacenar todos en una lista.

Ejemplos de Ejecución

$3 + 5 * (10 - 4)$

```

6      ('MINUS', r'-'),
7      ('MULT', r'\*'),
8      ('DIV', r'/'),
9      ('LPAREN', r'\('),
10     ('RPAREN', r'\)'),

('PLUS', '+')
('NUM', '6')
PS C:\Users\Familia\Desktop\tema 4 Lenguajes> & C:/Users/Familia/AppData/Local/Programs/Python/P
● /Lexer.py"
Ingrese una expresión aritmética: 3+5*(10-4)
Expresión ingresada: 3+5*(10-4)
('NUM', '3')
('PLUS', '+')
('NUM', '5')
('MULT', '*')
('LPAREN', '(')
('NUM', '10')
('MINUS', '-')
('NUM', '4')
('RPAREN', ')')
○ PS C:\Users\Familia\Desktop\tema 4 Lenguajes>

```

$7 * 10 / 5 (56 + 4)$


```

PS C:\Users\Familia\Desktop\tema 4 Lenguajes> & C:/Users/Famili
• /Lexer.py"
Ingrese una expresión aritmética: 7*10/5(56+4)
Expresión ingresada: 7*10/5(56+4)
('NUM', '7')
('MULT', '*')
('NUM', '10')
('DIV', '/')
('NUM', '5')
('LPAREN', '(')
('NUM', '56')
('PLUS', '+')
('NUM', '4')
('RPAREN', ')')
PS C:\Users\Familia\Desktop\tema 4 Lenguajes>

```

$7 * 7 / 5 / 6 + 55 - 6$

```

• /Lexer.py"
Ingrese una expresión aritmética: 7*7/5/6+55-6
Expresión ingresada: 7*7/5/6+55-6
('NUM', '7')
('MULT', '*')
('NUM', '7')
('DIV', '/')
('NUM', '5')
('DIV', '/')
('NUM', '6')
('PLUS', '+')
('NUM', '55')
('MINUS', '-')
('NUM', '6')
PS C:\Users\Familia\Desktop\tema 4 Lenguajes>

```

Este lexer es capaz de analizar expresiones aritméticas básicas y dividirlos en tokens utilizando expresiones regulares. Es un paso fundamental en la construcción de un compilador o intérprete para lenguajes que soporten operaciones matemáticas.

3. Definido un lenguaje L construya un lexer utilizando metacompilador

Un metacompilador es una herramienta diseñada para desarrollar compiladores u otras aplicaciones capaces de procesar código. En términos simples, actúa como un "compilador de compiladores", permitiendo la generación de programas que pueden analizar e interpretar distintos lenguajes o textos.

Flex es un metacompilador especializado en la construcción de analizadores léxicos, los cuales se encargan de descomponer un texto en sus unidades fundamentales, conocidas como tokens. Estos tokens pueden utilizarse en procesos más complejos, como la compilación de un lenguaje de programación.

3.1. Manual de usuario del metacompilador Flex

Flex (Fast Lexical Analyzer Generator) es una herramienta utilizada para desarrollar programas que pueden analizar y procesar texto, dividiéndolo en fragmentos más pequeños llamados tokens. Su función principal es generar código en C que identifique estos tokens de acuerdo con las reglas establecidas en un archivo de configuración.

Este archivo de configuración, generalmente con la extensión .l, emplea expresiones regulares para definir los patrones de texto que deben ser detectados. Además, sigue una estructura específica que permite a Flex interpretar y procesar correctamente el contenido.

```
Patron1 { acción 1 }
```

```
Patron2 { acción 2 }
```

Donde el patrón es una expresión regular y la acción es código C con las acciones

a. Estructura de un archivo Flex

Un archivo Flex se organiza en tres secciones, delimitadas por `{` y `}` en la primera parte y por `%%` en las otras dos.

1. Sección de Definiciones:

Aquí se pueden declarar macros y configuraciones, utilizando la sintaxis estándar de C. Aunque esta sección es opcional, es útil para mantener el código más estructurado.

```
%  
  
{ #include <stdlib.h>  
  
% }
```

2. Sección de Reglas:

En esta parte se establecen las expresiones regulares junto con las acciones que se ejecutarán cuando se detecte una coincidencia. Cada regla sigue el formato:

```

patrón {acción}

%%

"hola" { printf("Saludo detectado: %s\n", yytext); }

[0-9]+ { printf("Número encontrado: %s\n", yytext); }

```

3. Sección de Código:

Se emplea para agregar código C adicional, como la función principal u otras funciones auxiliares.

```

%%
int main() {
    yylex();
    return 0;
}

```

b. Patrones en Flex

Al trabajar con Flex, es fundamental definir cómo identificar las diferentes partes del texto a analizar. Esto se logra mediante la escritura de patrones, los cuales describen la estructura de los fragmentos de texto que se desean reconocer.

Por ejemplo, un patrón puede especificar que se detecten todos los números o todas las palabras que comiencen con una letra mayúscula.

Uso de Expresiones Regulares

Los patrones en Flex se representan mediante expresiones regulares, un lenguaje especializado que permite describir estructuras de texto con gran precisión. Estas expresiones actúan como un conjunto de reglas que le indican a Flex qué debe buscar dentro del texto de entrada.

Dentro de los patrones, se pueden utilizar cualquier carácter ASCII, incluyendo letras (A-Z, a-z), números (0-9) y símbolos como !, @, #, etc. Sin embargo, hay ciertos caracteres especiales que tienen funciones específicas en las expresiones regulares. Estos incluyen:

```

"" \ [ ^ - ? . * + | ( ) $ / { } % < >

```

Los espacios en blanco, tabulaciones y saltos de línea deben manejarse con cuidado, ya que no se incluyen directamente en los patrones.

Ejemplos de Patrones en Flex

[0-9]+ → Detecta cualquier secuencia de uno o más dígitos, como "123" o "7890".

[a-zA-Z]+ → Captura cualquier conjunto de letras, ya sean mayúsculas o minúsculas, como "hola" o "Programación".

[abj-oZ] → Identifica caracteres específicos dentro de un rango: la letra 'a', 'b', cualquier letra entre 'j' y 'o', o la 'Z'.

[^A-Z] → Encuentra cualquier carácter excepto las letras mayúsculas.

c. Emparejamiento de la entrada

Cuando Flex analiza el texto de entrada, su función es identificar las partes que coinciden con los patrones previamente definidos. Al encontrar una coincidencia, se dice que el patrón ha "emparejado" esa sección del texto.

El programa examina el texto en busca de fragmentos que coincidan con los patrones definidos en el archivo de Flex. Estos patrones pueden ser palabras específicas, números u otras secuencias de caracteres que se hayan especificado.

Si hay varias coincidencias posibles, Flex elegirá el patrón que coincida con la secuencia más larga. Por ejemplo, si existe un patrón que busca "rojo" y otro que busca "roj", y el texto contiene "rojo", el programa seleccionará "rojo" porque es el patrón más largo.

Si varios patrones coinciden con fragmentos de igual longitud, el programa opta por el que aparece primero en el archivo. Por ejemplo, si el texto contiene "perro" y los patrones son "perr" y "perro", el programa elegirá "perro" si aparece antes en el archivo.

Si el programa no encuentra ninguna coincidencia con los patrones definidos, sigue una regla predeterminada: toma el siguiente carácter del texto y lo copia tal cual a la salida. De esta forma, el carácter se procesa, aunque no se ajuste a ningún patrón específico.

d. Analizadores

Al ejecutar el programa, Flex genera un archivo llamado `lex.yy.c`, que contiene todo lo necesario para realizar el análisis del texto de entrada de acuerdo con las reglas definidas previamente. En este archivo se incluye una función principal llamada `yylex()`, que Flex crea automáticamente para encargarse de leer y procesar el texto. Cada vez que se invoca `yylex()`, esta función busca los patrones especificados en el texto y ejecuta las acciones correspondientes para cada uno.

Dentro de `lex.yy.c`, también se incluyen varias tablas que ayudan a la función `yylex()` a encontrar rápidamente qué partes del texto coinciden con los patrones que hemos definido. Además, el archivo cuenta con funciones adicionales y macros, que son fragmentos de código reutilizables, optimizando así el trabajo de `yylex()`.

Funcionamiento de `yylex()`

La función `yylex()` comienza leyendo el texto desde un archivo de entrada global denominado `yyin`. Por defecto, este archivo está configurado para leer desde la entrada estándar (`stdin`), lo que significa que puede tomar el texto ingresado por el usuario o el contenido de un archivo proporcionado como entrada.

Mientras lee, `yylex()` busca las coincidencias con los tokens definidos en los patrones. Cada vez que encuentra un token, ejecuta la acción asociada con el patrón correspondiente. El proceso continúa hasta que `yylex()` alcanza el final del archivo de entrada, momento en el cual devuelve el valor 0, indicando que ha terminado su tarea. También puede detenerse antes si alguna acción definida dentro del archivo contiene una instrucción `return`.

e. Compilación y Ejecución

Una vez que se ha escrito el archivo de Flex para el análisis de texto, el siguiente paso es compilar y ejecutar el programa generado.

Compilación del Archivo Flex

El primer paso es convertir el archivo Flex en un programa en C utilizando la herramienta Flex. Esto se logra mediante el comando:

```
flex nombre_fichero.l
```

Este comando genera un archivo llamado `lex.yy.c`, que contiene el código en C necesario para realizar el análisis, basado en los patrones y las acciones definidas en el archivo Flex.

Luego, se debe compilar el archivo `lex.yy.c` para convertirlo en un programa ejecutable. Esto se hace con el compilador GCC mediante el siguiente comando:

```
gcc lex.yy.c -lfl -o nombre_ejecutable
```

En este comando:

`gcc` es el compilador que se utiliza para compilar el código en C.

`lex.yy.c` es el archivo generado por Flex, que contiene el código en C.

`-lfl` le indica al compilador que debe enlazar el programa con la biblioteca de Flex, necesaria para que el análisis léxico funcione correctamente.

`-o nombre_ejecutable` especifica el nombre del archivo ejecutable que se generará.

Ejecución del Programa Compilado

Una vez que el programa ha sido compilado, puedes ejecutarlo directamente escribiendo el nombre del archivo ejecutable en la línea de comandos y presionando Enter:

```
./nombre_ejecutable
```

El programa comenzará a ejecutarse y estará esperando que ingreses texto. Cada vez que presiones Enter, el programa analizará el texto ingresado.

Si se desea probar el programa con entradas más largas, lo más sencillo es crear un archivo de texto y redirigirlo como entrada al programa. Esto se puede hacer utilizando redirección en la línea de comandos, de modo que el programa lea el contenido del archivo en lugar de solicitar entrada manualmente.

3.2. Descripción del Lenguaje L

Se creó un lenguaje de prueba, a continuación, podrá presenciar un resumen de su manual de usuario consolidado en tablas:

Básicos

Nombre	Símbolo	Uso
--------	---------	-----

DIGITO	[0-9]	Representa un solo dígito numérico.
LETRA	[a-zA-Z]	Representa una letra.
IDENTIFICADOR	{LETRA}({LETRA} {DIGITO})*	Define identificadores (nombres de variables, funciones, etc.).
NUMERO	{DIGITO}+	Representa un número entero.
CADENA	\".*\"	Representa un texto entre comillas.
COMA	,	separar parámetros en una función.
DOS_PUNTOS	:	definir bloques de código
PUNTOYCOMA	;	Marca el fin de la línea de código.

Identificador de Tipo de Dato

Nombre	Símbolo	Uso
TIPO_INT	int	Variable de tipo entero
TIPO_FLOAT	float	Variable de tipo float
TIPO_STRING	string	Cadena de texto

Palabras reservadas

Nombre	Símbolo	Uso
SI	si	Define una estructura condicional, similar a if en otros lenguajes. Se usa para ejecutar un bloque de código si una condición es verdadera.

SINO	sino	Se usa después de si para definir una alternativa cuando la condición del si no se cumple, similar a else.
MIENTRAS	mientras	Representa un bucle que se ejecuta mientras una condición sea verdadera, equivalente a while.
PARA	para	Define un bucle con inicialización, condición y actualización, similar a for. Se usa para iteraciones controladas.
ROMPER	romper	Detiene la ejecución de un bucle (mientras o para) antes de que la condición sea falsa, equivalente a break.
CONTINUAR	continuar	Salta la iteración actual de un bucle y pasa a la siguiente, similar a continue.
IMPRIMIR	imprimir	Imprimir en pantalla

Operadores de comparación

Nombre	Símbolo	Uso
OPERADOR_IG	==	Comprueba si dos valores son iguales.
OPERADOR_DIFF	!=	Verifica si dos valores son distintos.
OPERADOR_MAY	>	Indica si un valor es mayor que otro.
OPERADOR_MEN	<	Indica si un valor es menor que otro.
OPERADOR_MAYIG	>=	Verifica si un valor es mayor o igual.
OPERADOR_MENIG	<=	Verifica si un valor es menor o igual.

Operadores de aritméticos y asignación

Nombre	Símbolo	Uso
OPERADOR_SUM	+	Operador para la adición

OPERADOR_RES	-	Operador de sustracción
OPERADOR_MUL	*	Operador para multiplicación
OPERADOR_DIV	/	Operador de división o modulo
OPERADOR_ASIG	=	Operador de asignación

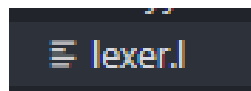
Símbolos de agrupación

Nombre	Símbolo	Uso
PARENTESIS_IZQ	"("	Inicia una expresión o parámetro.
PARENTESIS_DER	")"	Operador de Cierra una expresión o parámetro.
LLAVE_IZQ	"{"	Inicia un bloque de código.
LLAVE_DER	"}"	Cierra un bloque de código.

3.3. Proceso de creación de lexer con FLEX

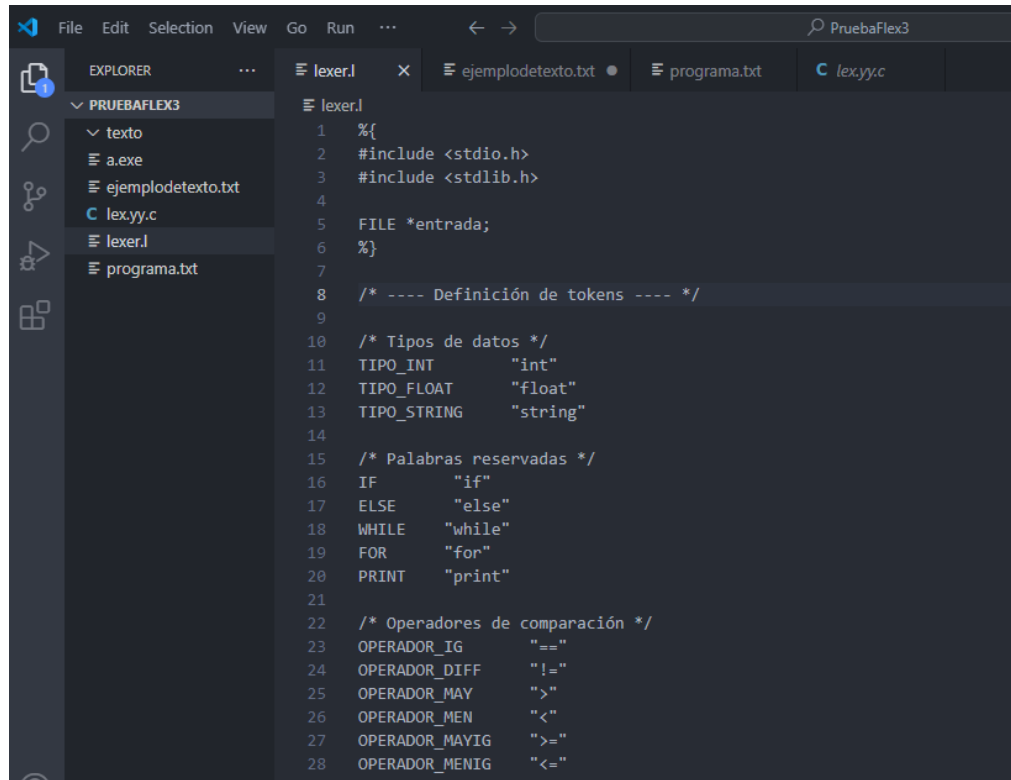
Para comenzar a utilizar el metacompilador FLEX, primero es indispensable contar con un compilador de código C/C++; en este caso, se empleará MinGW. Se procede a instalar FLEX y a configurar su ruta en las variables de entorno del sistema, de igual forma que se configura el directorio donde se instaló el compilador.

Posteriormente, es necesario revisar la documentación de FLEX para comprender su funcionamiento. Como entorno de desarrollo, se puede optar por Visual Studio Code y utilizar la línea de comandos (cmd o command prompt). Se crea una carpeta y, dentro de ella, se genera el primer archivo con extensión .l, al que se le asigna el nombre lexer.l, dado que se trata de un analizador léxico.



Tras haber revisado la documentación de FLEX, estamos listos para iniciar la construcción del lexer. Es fundamental establecer previamente los tokens y las reglas semánticas que definirán la estructura del lenguaje que se analizará, en este caso, el lenguaje L descrito en el informe.

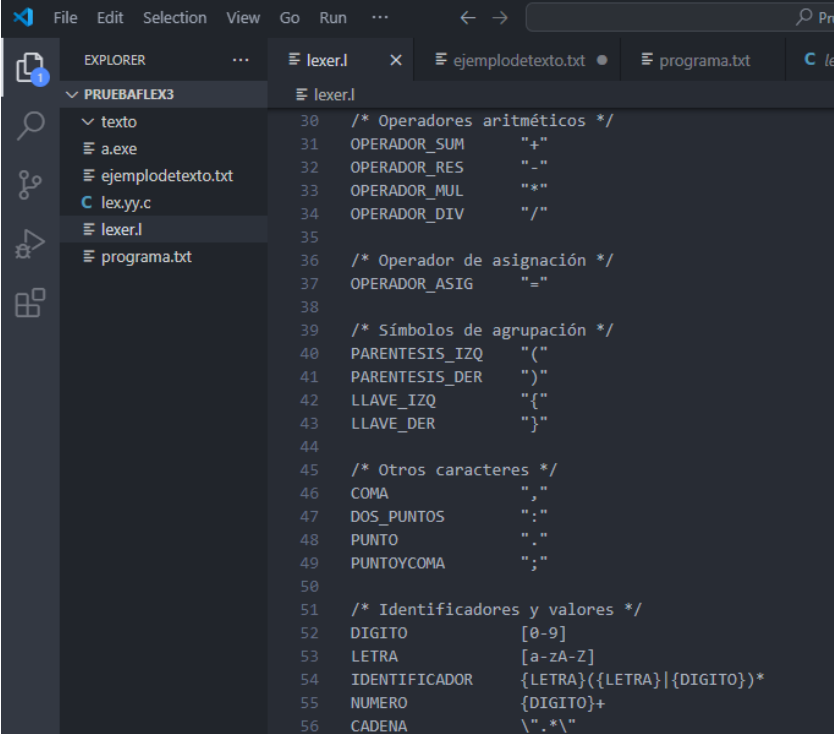
En la primera sección del código, se especifican los tokens junto con sus reglas gramaticales, lo que permite que el analizador determine a qué categoría pertenece cada token que recibe.



The screenshot shows a code editor with the following content in the `lexer.l` file:

```
1  %{
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  FILE *entrada;
6  %}
7
8  /* ---- Definición de tokens ---- */
9
10 /* Tipos de datos */
11 TIPO_INT      "int"
12 TIPO_FLOAT    "float"
13 TIPO_STRING   "string"
14
15 /* Palabras reservadas */
16 IF            "if"
17 ELSE          "else"
18 WHILE         "while"
19 FOR           "for"
20 PRINT         "print"
21
22 /* Operadores de comparación */
23 OPERADOR_IG   "=="
24 OPERADOR_DIFF "!="
25 OPERADOR_MAY  ">"
26 OPERADOR_MEN  "<"
27 OPERADOR_MAYIG ">="
28 OPERADOR_MENIG "<="
```

Se realiza la declaración de la variable `FILE *entrada` para se lea la entrada desde un archivo en lugar de la entrada estándar (`stdin`).



The screenshot shows a code editor with a dark theme. On the left, the Explorer panel shows a project named 'PRUEBAFLEX3' with a folder 'texto' containing files 'a.exe', 'ejemplodetexto.txt', 'lex.yy.c', 'lexer.l', and 'programa.txt'. The main editor window displays the content of 'lexer.l'. The code defines various tokens for a lexer, including arithmetic operators, assignment, grouping symbols, other characters, and identifiers/values.

```

30  /* Operadores aritméticos */
31  OPERADOR_SUM      "+"
32  OPERADOR_RES      "-"
33  OPERADOR_MUL      "*"
34  OPERADOR_DIV      "/"
35
36  /* Operador de asignación */
37  OPERADOR_ASIG     "="
38
39  /* Símbolos de agrupación */
40  PARENTESIS_IZQ    "("
41  PARENTESIS_DER    ")"
42  LLAVE_IZQ         "{"
43  LLAVE_DER         "}"
44
45  /* Otros caracteres */
46  COMA               ","
47  DOS_PUNTOS         ":"
48  PUNTO              "."
49  PUNTOYCOMA         ";"
50
51  /* Identificadores y valores */
52  DIGITO             [0-9]
53  LETRA              [a-zA-Z]
54  IDENTIFICADOR      {LETRA}{(LETRA|DIGITO)}*
55  NUMERO             {DIGITO}+
56  CADENA             "\".*\""

```

Esta sección define qué hacer cuando se detecta cada token. Cada vez que el lexer reconoce int, float o string, imprime el tipo de token detectado y su lexema (yytext contiene el texto reconocido).

```

56 CADENA          \".*\\"
57
58 /* ---- Reglas de reconocimiento ---- */
59
60 %%
61
62 {TIPO_INT}        { printf("TIPO_INT: %s\n", yytext); }
63 {TIPO_FLOAT}      { printf("TIPO_FLOAT: %s\n", yytext); }
64 {TIPO_STRING}     { printf("TIPO_STRING: %s\n", yytext); }
65
66 {IF}              { printf("PALABRA RESERVADA IF: %s\n", yytext); }
67 {ELSE}            { printf("PALABRA RESERVADA ELSE: %s\n", yytext); }
68 {WHILE}           { printf("PALABRA RESERVADA WHILE: %s\n", yytext); }
69 {FOR}             { printf("PALABRA RESERVADA FOR: %s\n", yytext); }
70 {PRINT}           { printf("PALABRA RESERVADA PRINT: %s\n", yytext); }
71
72 {OPERADOR_IG}     { printf("OPERADOR_IG: %s\n", yytext); }
73 {OPERADOR_DIFF}   { printf("OPERADOR_DIFF: %s\n", yytext); }
74 {OPERADOR_MAY}    { printf("OPERADOR_MAY: %s\n", yytext); }
75 {OPERADOR_MEN}    { printf("OPERADOR_MEN: %s\n", yytext); }
76 {OPERADOR_MAYIG}  { printf("OPERADOR_MAYIG: %s\n", yytext); }
77 {OPERADOR_MENIG}  { printf("OPERADOR_MENIG: %s\n", yytext); }
78
79 {OPERADOR_SUM}    { printf("OPERADOR_SUM: %s\n", yytext); }
80 {OPERADOR_RES}    { printf("OPERADOR_RES: %s\n", yytext); }
81 {OPERADOR_MUL}    { printf("OPERADOR_MUL: %s\n", yytext); }
82 {OPERADOR_DIV}    { printf("OPERADOR_DIV: %s\n", yytext); }
83
84 {OPERADOR_ASIG}   { printf("OPERADOR_ASIG: %s\n", yytext); }
85

```

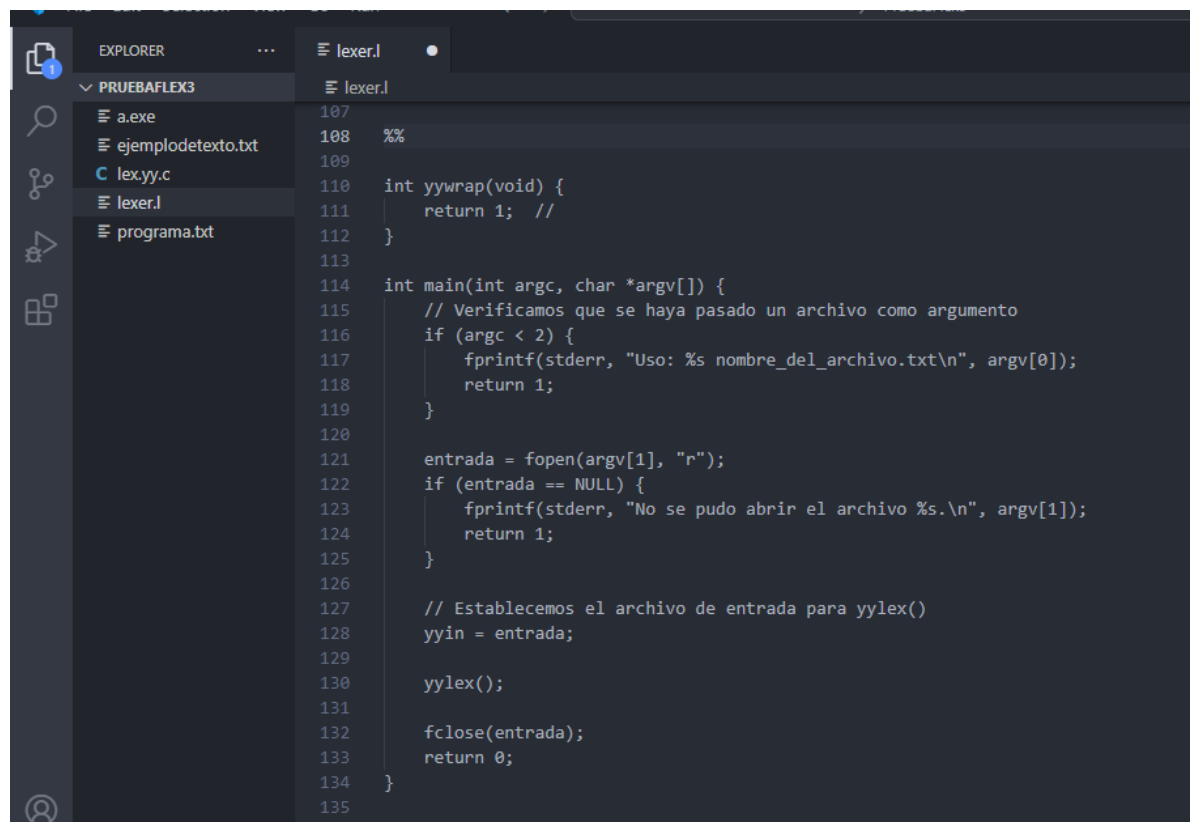
```

86 {OPERADOR_DIV}    { printf("OPERADOR_DIV: %s\n", yytext); }
87
88 {OPERADOR_ASIG}   { printf("OPERADOR_ASIG: %s\n", yytext); }
89
90 {PARENTESIS_IZQ}  { printf("PARENTESIS_IZQ: %s\n", yytext); }
91 {PARENTESIS_DER}  { printf("PARENTESIS_DER: %s\n", yytext); }
92 {LLAVE_IZQ}       { printf("LLAVE_IZQ: %s\n", yytext); }
93 {LLAVE_DER}       { printf("LLAVE_DER: %s\n", yytext); }
94
95 {COMA}            { printf("COMA: %s\n", yytext); }
96 {DOS_PUNTOS}      { printf("DOS_PUNTOS: %s\n", yytext); }
97 {PUNTO}           { printf("PUNTO: %s\n", yytext); }
98 {PUNTOYCOMA}      { printf("PUNTOYCOMA: %s\n", yytext); }
99
100 {NUMERO}          { printf("NUMERO: %s\n", yytext); }
101 {CADENA}          { printf("CADENA: %s\n", yytext); }
102 {IDENTIFICADOR}   { printf("IDENTIFICADOR: %s\n", yytext); }
103
104 [ \t\n]           { /* Ignorar espacios, tabulaciones y saltos de línea */ }
105
106 "/*".*           { /* Comentario de una línea, ignorado */ }
107 "/*"[^"]*" */    { /* Comentario de varias líneas, ignorado */ }
108
109 .                { printf("CARACTER INVALIDO: %s\n", yytext); }
110
111 %%

```

El código en C y Flex implementa un analizador léxico que procesa un archivo de entrada. La función `yywrap()` es utilizada por Flex y se ejecuta cuando `yylex()` llega al final del archivo. Su propósito es indicar si hay más archivos por procesar, pero en este caso, al retornar 1, señala que el análisis ha finalizado. Se hizo uso de esta función debido a que en algunos sistemas la biblioteca no está disponible o no se enlaza automáticamente cuando se instala Flex entonces se define manualmente para no ocasionar errores si no se encuentra en la biblioteca externa.

La función `main()` es la principal del programa y comienza verificando si el usuario ha proporcionado un archivo como argumento en la línea de comandos. Si no se especifica un archivo, se muestra un mensaje de error y el programa termina.



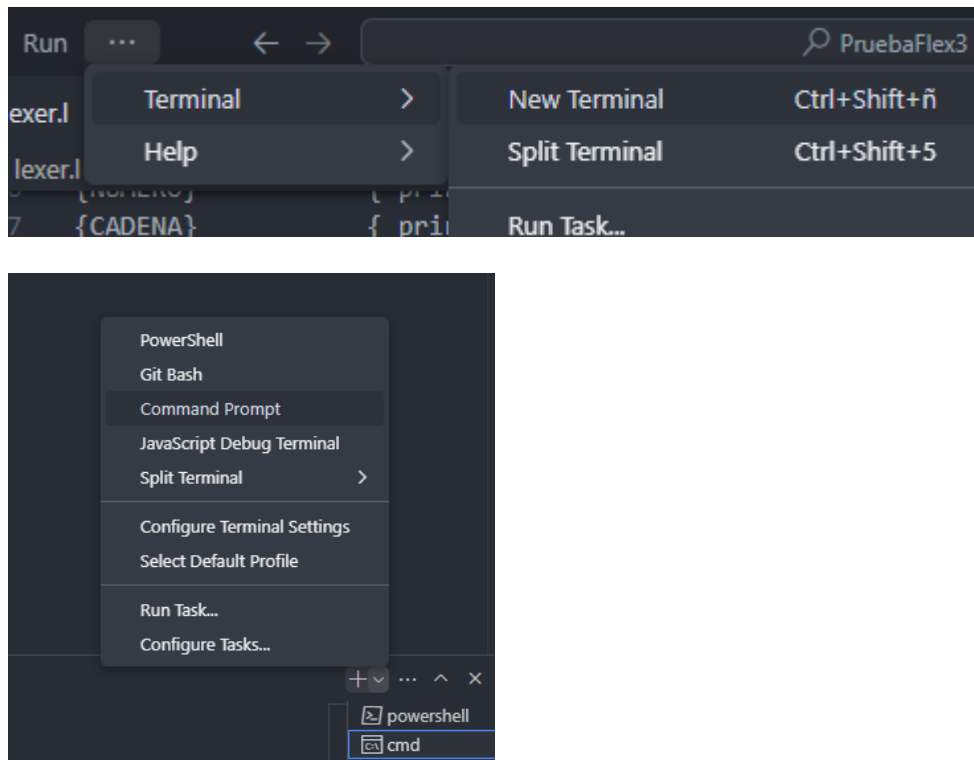
```

107
108 %%
109
110 int yywrap(void) {
111     return 1; //
112 }
113
114 int main(int argc, char *argv[]) {
115     // Verificamos que se haya pasado un archivo como argumento
116     if (argc < 2) {
117         fprintf(stderr, "Uso: %s nombre_del_archivo.txt\n", argv[0]);
118         return 1;
119     }
120
121     entrada = fopen(argv[1], "r");
122     if (entrada == NULL) {
123         fprintf(stderr, "No se pudo abrir el archivo %s.\n", argv[1]);
124         return 1;
125     }
126
127     // Establecemos el archivo de entrada para yylex()
128     yyin = entrada;
129
130     yylex();
131
132     fclose(entrada);
133     return 0;
134 }
135

```

Si el archivo se abre correctamente, se asigna a `yyin`, una variable global utilizada por Flex para definir la fuente de entrada del análisis léxico, se hace con `(stdin)` por defecto, pero en esta ocasión estamos usando el lexer de manera que lea cualquier archivo que le pasemos como argumento. Luego, se llama a `yylex()`, la función generada por Flex encargada de leer el contenido del archivo y reconocer los tokens definidos en las reglas del analizador léxico.

Una vez que tenemos listo el código del analizador léxico, abrimos un terminal puede ser powershell o command prompt



Para convertir nuestro archivo **.l** en código C usamos Flex ejecutando de la siguiente manera:

```
C:\Users\User\Documents\Lenguajes Metacompilador\PruebaFlex3>flex lexer.l
```

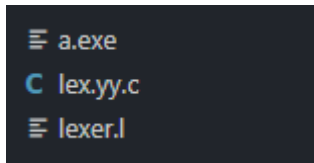
Esto genera un archivo llamado **lex.yy.c**, que es el código fuente en C del analizador léxico.

```
C lex.yy.c
lex.l
```

Luego, usamos el compilador de C para compilar el lexer generado

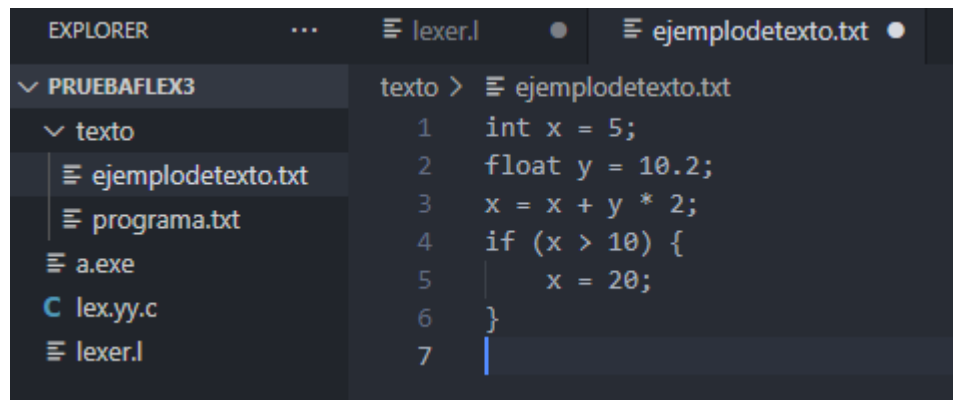
```
C:\Users\User\Documents\Lenguajes Metacompilador\PruebaFlex3>gcc lex.yy.c
```

Aquí, gcc traduce el código a un ejecutable llamado **a.exe**



De esta manera hemos dispuesto el analizador para que pueda leer cualquier archivo que le pasemos.

Ya hemos creado un archivo y lo hemos denominado ejemplodetexto.txt, este va a contener un fragmento de código en lenguaje L el cual podremos analizar.



Con el ejecutable que hemos obtenido anteriormente (a.exe), vamos a abrir el archivo donde tenemos nuestro fragmento de código en lenguaje L

```

C:\Users\User\Documents\Lenguajes Metacompilador\PruebaFlex3>a.exe ejemplodetexto.txt
TIPO_INT: int
IDENTIFICADOR: x
OPERADOR_ASIG: =
NUMERO: 5
PUNTOYCOMA: ;
TIPO_FLOAT: float
IDENTIFICADOR: y
OPERADOR_ASIG: =
NUMERO: 10
PUNTO: .
NUMERO: 2
PUNTOYCOMA: ;
IDENTIFICADOR: x
OPERADOR_ASIG: =
IDENTIFICADOR: x
OPERADOR_SUM: +
IDENTIFICADOR: y
OPERADOR_MUL: *
NUMERO: 2
PUNTOYCOMA: ;
PALABRA RESERVADA IF: if
PARENTESIS_IZQ: (
IDENTIFICADOR: x
OPERADOR_MAY: >
NUMERO: 10
PARENTESIS_DER: )
LLAVE_IZQ: {
IDENTIFICADOR: x
OPERADOR_ASIG: =
NUMERO: 20
PUNTOYCOMA: ;
LLAVE_DER: }

```

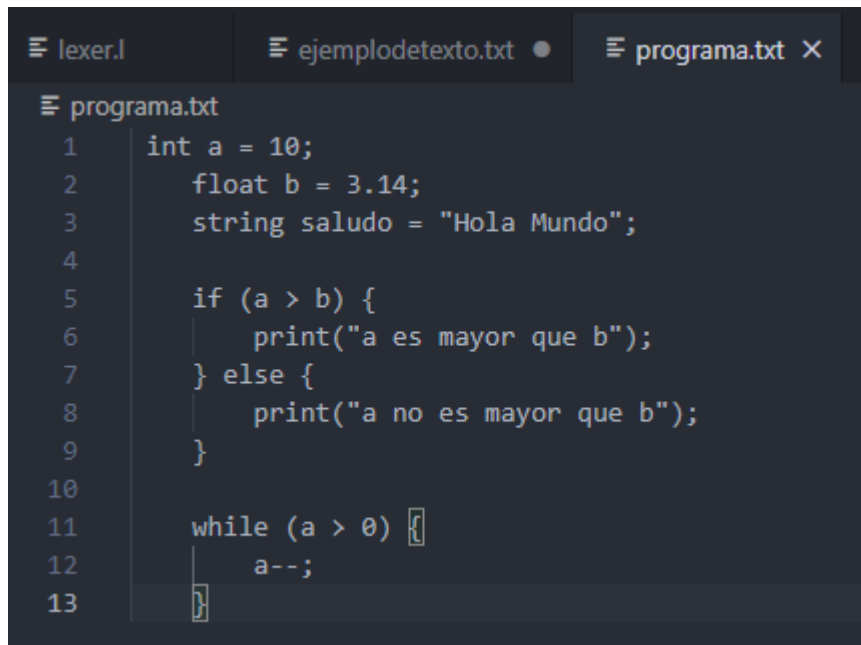
```

PUNTOYCOMA: ;
LLAVE_DER: }
PALABRA RESERVADA IF: if
IDENTIFICADOR: hola
OPERADOR_IG: ==
NUMERO: 3

```

Como se puede observar, el archivo se ha abierto y leído correctamente, realizando un análisis detallado de cada token conforme a las reglas definidas previamente. Una vez generado el ejecutable, es posible utilizarlo para procesar otros archivos y continuar evaluando el funcionamiento del analizador léxico desarrollado con FLEX.

Probaremos ahora el ejecutable con el archivo programa.txt:



```
lexer.l  ejemplodetexto.txt  programa.txt X
programa.txt
1  int a = 10;
2      float b = 3.14;
3      string saludo = "Hola Mundo";
4
5      if (a > b) {
6          print("a es mayor que b");
7      } else {
8          print("a no es mayor que b");
9      }
10
11     while (a > 0) {
12         a--;
13     }
```

Ejecutamos en el terminal command prompt: a.exe programa.txt

```
C:\Users\User\Documents\Lenguajes Metacompilador\PruebaFlex3>a.exe programa.txt
TIPO_INT: int
IDENTIFICADOR: a
OPERADOR_ASIG: =
NUMERO: 10
PUNTOYCOMA: ;
TIPO_FLOAT: float
IDENTIFICADOR: b
OPERADOR_ASIG: =
NUMERO: 3
PUNTO: .
NUMERO: 14
PUNTOYCOMA: ;
TIPO_STRING: string
IDENTIFICADOR: saludo
OPERADOR_ASIG: =
CADENA: "Hola Mundo"
PUNTOYCOMA: ;
PALABRA RESERVADA IF: if
PARENTESIS_IZQ: (
IDENTIFICADOR: a
OPERADOR_MAY: >
IDENTIFICADOR: b
PARENTESIS_DER: )
LLAVE_IZQ: {
PALABRA RESERVADA PRINT: print
PARENTESIS_IZQ: (
CADENA: "a es mayor que b"
PARENTESIS_DER: )
```

```
IDENTIFICADOR: b
PARENTESIS_DER: )
LLAVE_IZQ: {
PALABRA RESERVADA PRINT: print
PARENTESIS_IZQ: (
CADENA: "a es mayor que b"
PARENTESIS_DER: )
PUNTOYCOMA: ;
LLAVE_DER: }
PALABRA RESERVADA ELSE: else
LLAVE_IZQ: {
PALABRA RESERVADA PRINT: print
PARENTESIS_IZQ: (
CADENA: "a no es mayor que b"
PARENTESIS_DER: )
PUNTOYCOMA: ;
LLAVE_DER: }
PALABRA RESERVADA WHILE: while
PARENTESIS_IZQ: (
IDENTIFICADOR: a
OPERADOR_MAY: >
NUMERO: 0
PARENTESIS_DER: )
LLAVE_IZQ: {
IDENTIFICADOR: a
OPERADOR_RES: -
OPERADOR_RES: -
PUNTOYCOMA: ;
LLAVE_DER: }
```

Referencias Bibliográficas

- Fernández, A. (2023). Desarrollo de un lexer para lenguaje de programación educativo [Tesis de maestría, Universidad de Buenos Aires]. Repositorio Digital UBA. Recuperado de <https://www.uba.ar/repositorio>
- Morales, E. (2022). *Generación automática de analizadores léxicos con herramientas de código abierto* [Tesis doctoral, Universidad de Sevilla]. Repositorio US. Recuperado de <https://www.us.es/repositorio>
- Jiménez, C. (2021). *Flex y Bison: Aplicación en la enseñanza de compiladores* [Tesis de licenciatura, Universidad Nacional de Colombia]. Biblioteca Digital UNAL. Recuperado de <https://www.bdigital.unal.edu.co>
- Universidad Nacional Autónoma de México (UNAM). (2025). *Manual de uso de Flex y Bison para análisis léxico y sintáctico*. Recuperado de <https://www.unam.mx/docs/flexbison>
- Vargas, L., & Méndez, R. (2020). *Diseño e implementación de un lexer basado en expresiones regulares*. *Revista Latinoamericana de Computación*, 18(2), 45-60. Recuperado de <https://www.revistalatinoamericanadecomputacion.com>
- López, J., & Gómez, M. (2021). *Análisis léxico y sintáctico: Fundamentos para compiladores modernos*. Editorial Pearson. Recuperado de <https://www.pearson.com/es>
- Hopcroft, J. E., & Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. Addison-Wesley.