UNIVERSIDAD DEL VALLE DE GUATEMALA

CC3071- Diseño de lenguajes de programación

Sección 31

Laboratorio 1

Jose Auyón, 201579

GUATEMALA, 15 de febrero del 2025

Link de github: https://github.com/auyjos/Flex_Setup_Jose

Repaso conceptos:

1. Defina qué hace la función retractar dentro del proceso de reconocimiento de tokens. Explique:

- En qué momento se invoca esta función.

retract() deshace la última lectura del buffer: retrocede el puntero de entrada uno (o más) caracteres para "devolver" caracteres que se leyeron de más mientras el autómata intentaba extender el lexema. Se invoca justo cuando el analizador se da cuenta de que ya no puede seguir avanzando (no hay transición válida para el carácter actual), pero ya había alcanzado un estado de aceptación antes. Es decir: se leyó un carácter extra que *no pertenece* al token, y antes de devolver el token hay que "regresar" ese carácter para que sea procesado por el siguiente token.

- Qué sucede con el puntero del buffer al ejecutarse.

El puntero (índice) del buffer retrocede al carácter anterior (o a la posición del último carácter que NO debía consumirse). En implementaciones: Si se lee un carácter c que rompió el patrón, hace retract() para que c quede como "siguiente carácter" para el siguiente token. Algunas implementaciones guardan "última posición aceptada" y retroceden hasta ahí.

- Por qué es necesaria para aceptar correctamente un token.

Porque en análisis léxico se usa máxima coincidencia: el lexer intenta consumir lo más posible. Eso implica que con frecuencia necesita mirar un carácter más para decidir que el token terminó. Sin retract, ese carácter se perdería o quedaría mal asignado.

Ejemplo: entrada abc+12 con ID = letra (letra|digito)*

- El lexer lee a b c y está aceptando ID.
- Lee + y el DFA de ID ya no tiene transición.
- Ese + no es parte del ID, entonces haces retract() y devuelves ID("abc"). Luego el + se analiza como otro token.

- Cómo influye en el comportamiento del autómata finito.

El DFA no “retrocede” . El retract no es del DFA, es del motor del lexer: controla el cursor de entrada para que el DFA pueda avanzar con para lograr máxima coincidencia, y luego “corregir” el consumo cuando se excede. Sin retract, el DFA terminaría aceptando tokens más cortos o consumiría caracteres que pertenecen al siguiente token, rompiendo la segmentación del input.

2. Defina que hace la función fallo en el contexto del análisis léxico. Explique:

fallo () se ejecuta cuando el patrón/token actual no puede reconocer la entrada desde la posición inicial. En vez de declarar error inmediatamente, el lexer intenta otro patrón.

- Qué significa que un autómata "falle".

Que, empezando desde el estado inicial de ese token, para el primer carácter (o en algún punto) no existe transición válida y además nunca se llegó a un estado de aceptación. O sea: ese patrón no aplica aquí.

- Qué ocurre cuando no existe una transición válida para el carácter leído.

Hay dos escenarios:

- Ya hubo aceptación previa: entonces NO es “fallo”; es fin de token → devuelves token y probablemente retractas.
- Nunca hubo aceptación: entonces sí es “fallo” del patrón → se llama fallo().

- Cómo permite esta función intentar el reconocimiento de otro patrón.

El lexer típicamente mantiene una lista/orden de patrones: oprel, ID, NUM, WS, etc.
Si oprel falla en la posición actual, fallo() hace que el lexer pruebe ID; si ID falla, prueba NUM, etc., sin mover el puntero inicial, hasta que alguno reconozca algo.

- Su relación con la estrategia de máxima coincidencia.

La máxima coincidencia dice: “elige el token que consume más caracteres”. Para lograrlo, el motor suele y se suele probar patrones, medir cuál llegó más lejos en aceptación, y desempatar por precedencia.

fallo() es parte del mecanismo de exploración: cuando un patrón no puede ni arrancar, lo descartas rápido para evaluar otros y quedarte con el que mejor coincide.

3. Defina la función error dentro del proceso de análisis léxico. Su explicación debe incluir:

- En qué situación se produce un error léxico.

Cuando el carácter (o secuencia) actual no pertenece al alfabeto válido del lenguaje, o viola reglas básicas de tokens. Ejemplos:

- símbolo ilegal: @ si no existe token que lo acepte,
- número mal formado: 12. si tu gramática exige dígitos después del punto,
- exponente incompleto: 1E+ sin dígitos.

- Qué ocurre cuando ningún autómata reconoce la secuencia de entrada.

El lexer no puede producir token, entonces:

- reporta un error con posición (línea, columna),
- decide cómo recuperarse para no quedarse en bucle.

    - Cómo se reporta el error.

Normalmente se reporta:

- el carácter inesperado o fragmento,
- la posición exacta (línea/columna),
- un mensaje de contexto (“símbolo no reconocido”, “número mal formado”, etc.).

Ejemplo de mensaje:

> Error léxico en línea 3, columna 15: carácter inesperado '@'

    - Qué mecanismos de recuperación pueden aplicarse.

Depende del diseño, pero lo típico:

1. **Panic mode:** consumir 1 carácter y seguir intentando (muy común).
2. **Saltar hasta delimitador:** consumir hasta espacio, ;, salto de línea, etc.
3. **Insertar/omitir:** en casos específicos. Lo mínimo para que funcione: avanzar el puntero al menos 1 carácter.

4. Provea un ejemplo como el que se vió en clase.

oprel → < | > | <= | >= | = | <>

ID → letra (letra|digito)*

NUM → digito+ (. digito+)? (E (+|−)? digito+)?

WS → space+

Entrada: total<=12.5E-2 @ x

Paso	char	state	lexeb	found	Action
1	t	—	t	t	fallo(oprel)

2	t	10	t	t	reconocer(t)
3	o	10	t	o	reconocer(o)
4	t	10	t	t	reconocer(t)
5	a	10	t	a	reconocer(a)
6	l	10	t	l	reconocer(l)
7	<	11	t	<	retract(<)
8	—	11	t	<	match(ID = "total")
9	<	1	<	<	reconocer(<)
10	=	2	<	=	reconocer(=)
11	—	2	<	=	match(oprel = "<=")
12	1	—	1	1	fallo(oprel)
13	1	—	1	1	fallo(ID)
14	1	12	1	1	reconocer(1)
15	2	13	1	2	reconocer(2)
16	.	16	1	.	reconocer(.)
17	5	18	1	5	reconocer(5)

18	**E**	**17**	**1**	**E**	**reconocer(E)**
19	**-**	**21**	**1**	**-**	**reconocer(-)**
20	**2**	**15**	**1**	**2**	**reconocer(2)**
21	**(space)**	**14**	**1**	**(space)**	**retract(space)**
22	**—**	**14**	**1**	**(space)**	**match(NUM = "12.5E-2")**
23	**(space)**	**23**	**(space)**	**(space)**	**reconocer(space)**
24	**@**	**24**	**(space)**	**@**	**retract(@)**
25	**—**	**24**	**(space)**	**@**	**match(WS) *(skip)***
26	**@**	**—**	**@**	**@**	**fallo(oprel)**
27	**@**	**—**	**@**	**@**	**fallo(ID)**
28	**@**	**—**	**@**	**@**	**fallo(NUM)**
29	**@**	**—**	**@**	**@**	**fallo(WS)**
30	**—**	**—**	**@**	**@**	**error('@') *(reportar y consumir '@')***
31	**(space)**	**23**	**(space)**	**(space)**	**reconocer(space)**

32	**x**	**24**	**(space)**	**x**	**retract(x)**
33	**—**	**24**	**(space)**	**x**	**match(WS) *(skip)***
34	**x**	**—**	**x**	**x**	**fallo(oprel)**
35	**x**	**10**	**x**	**x**	**reconocer(x)**
36	**—**	**10**	**x**	**EOF**	**match(ID = "x")**

2) **Actividades de Experimentación con Flex:**

1. Cree un archivo de especificación Flex (.l) que reconozca identificadores válidos en Java (que comiencen con letra o guion bajo, seguidos de letras, dígitos o guiones bajos). Compile y pruebe con diferentes casos de prueba.

2. Implemente un lexer que reconozca y clasifique los diferentes tipos de literales numéricos: enteros, flotantes, notación científica, y números hexadecimales. Para cada tipo, el lexer debe imprimir el token y su valor.

3. Extienda su lexer para reconocer operadores aritm´eticos (+, -, *, /), relacionales (==, !=, <, >, <=, >=), y lógicos (&&, ||, !).

4. Agregue soporte para reconocer comentarios de una l´ınea (//) y multil´ınea (/* */). Los comentarios deben ser descartados por el lexer y no generar tokens.

5. Implemente el reconocimiento de cadenas literales entre comillas dobles, manejando correctamente secuencias de escape como \n, \t, \", y \\.

6. Realice pruebas en otro lenguaje de su elección, que no sea python ni java.

```
root@e3c89dfa6ac3:/workspace# cd examples/
root@e3c89dfa6ac3:/workspace/examples# ./run_tests.sh
==========================================
  Compilando java_lexer.l con Flex
==========================================
  ✓ Compilación exitosa

==========================================
  PRUEBA 1: Archivo Java (test_java.txt)
==========================================
==========================================
  Lexer de Java — Análisis Léxico con Flex
==========================================

Archivo: test_java.txt

  [COMMENT_SINGLE_LINE descartado] line=1
  [COMMENT_SINGLE_LINE descartado] line=2
  [COMMENT_SINGLE_LINE descartado] line=3
  [COMMENT_SINGLE_LINE descartado] line=4
  [COMMENT_SINGLE_LINE descartado] line=6
Token(IDENTIFIER          , 'int', line=7, col=1)
Token(IDENTIFIER          , 'miVariable', line=7, col=5)
Token(ASSIGN              , '=', line=7, col=16)
Token(INTEGER_LITERAL     , '10', line=7, col=18)
Token(SEMICOLON           , ';', line=7, col=20)
Token(IDENTIFIER          , 'String', line=8, col=1)
Token(IDENTIFIER          , '_nombre', line=8, col=8)
Token(ASSIGN              , '=', line=8, col=16)
Token(STRING_LITERAL      , '"Hola"', line=8, col=18)
Token(SEMICOLON           , ';', line=8, col=24)
Token(IDENTIFIER          , 'double', line=9, col=1)
Token(IDENTIFIER          , '__valor123', line=9, col=8)
Token(ASSIGN              , '=', line=9, col=19)
Token(FLOAT_LITERAL       , '3.14', line=9, col=21)
Token(SEMICOLON           , ';', line=9, col=25)
Token(IDENTIFIER          , 'int', line=10, col=1)
Token(IDENTIFIER          , 'camelCase', line=10, col=5)
Token(ASSIGN              , '=', line=10, col=15)
Token(INTEGER_LITERAL     , '1', line=10, col=17)
Token(SEMICOLON           , ';', line=10, col=18)
Token(IDENTIFIER          , 'int', line=11, col=1)
Token(IDENTIFIER          , 'PascalCase', line=11, col=5)
Token(ASSIGN              , '=', line=11, col=16)
Token(INTEGER_LITERAL     , '2', line=11, col=18)
Token(SEMICOLON           , ';', line=11, col=19)
Token(IDENTIFIER          , 'int', line=12, col=1)
```

```
==========================================
  PRUEBA 2: Archivo Go (test_go.go)
==========================================
==========================================
  Lexer de Java — Análisis Léxico con Flex
==========================================

Archivo: test_go.go

  [COMMENT_SINGLE_LINE descartado] line=1
  [COMMENT_SINGLE_LINE descartado] line=2
  [COMMENT_SINGLE_LINE descartado] line=3
  [COMMENT_SINGLE_LINE descartado] line=4
  [COMMENT_SINGLE_LINE descartado] line=5
Token(IDENTIFIER          , 'package', line=7, col=1)
Token(IDENTIFIER          , 'main', line=7, col=9)
Token(IDENTIFIER          , 'import', line=9, col=1)
Token(LPAREN              , '(', line=9, col=8)
Token(STRING_LITERAL      , '"fmt"', line=10, col=2)
Token(RPAREN              , ')', line=11, col=1)
  [COMMENT_SINGLE_LINE descartado] line=13
Token(IDENTIFIER          , 'const', line=14, col=1)
Token(IDENTIFIER          , 'PI', line=14, col=7)
Token(ASSIGN              , '=', line=14, col=10)
Token(FLOAT_LITERAL       , '3.14159265', line=14, col=12)
Token(IDENTIFIER          , 'const', line=15, col=1)
Token(IDENTIFIER          , 'MAX_SIZE', line=15, col=7)
Token(ASSIGN              , '=', line=15, col=16)
Token(INTEGER_LITERAL     , '1000', line=15, col=18)
Token(IDENTIFIER          , 'const', line=16, col=1)
Token(IDENTIFIER          , 'HEX_COLOR', line=16, col=7)
Token(ASSIGN              , '=', line=16, col=17)
Token(HEX_INTEGER         , '0xFF00AB', line=16, col=19)
Token(IDENTIFIER          , 'const', line=17, col=1)
Token(IDENTIFIER          , 'AVOGADRO', line=17, col=7)
Token(ASSIGN              , '=', line=17, col=16)
Token(SCIENTIFIC_FLOAT    , '6.022e23', line=17, col=18)
  [COMMENT_MULTI_LINE descartado] line=19
Token(IDENTIFIER          , 'func', line=24, col=1)
Token(IDENTIFIER          , 'calcularArea', line=24, col=6)
Token(LPAREN              , '(', line=24, col=18)
Token(IDENTIFIER          , 'radio', line=24, col=19)
Token(IDENTIFIER          , 'float64', line=24, col=25)
Token(RPAREN              , ')', line=24, col=32)
Token(IDENTIFIER          , 'float64', line=24, col=34)
Token(LBRACE              , '{', line=24, col=42)
Token(IDENTIFIER          , 'return', line=25, col=2)
```

7. Explique que tendría que hacer si usted quisiera cambiar el color de algunas palabras

(al estilo visual studio code)

Para lograr un resaltado de sintaxis como el de Visual Studio Code, no basta con “pintar texto” desde el lexer. En editores modernos, el color es la etapa final de un pipeline: primero se clasifica el texto (tokenización), luego se etiqueta con categorías estándar, y al final el tema (theme) decide el color de cada categoría. En VS Code, el mecanismo más común para esto es TextMate: se define una gramática (normalmente en un archivo .tmLanguage.json) con expresiones regulares que detectan comentarios, cadenas, números, palabras reservadas, operadores e identificadores. Cada patrón se etiqueta con un scope (por ejemplo, keyword.control, constant.numeric, string.quoted.double, comment.line). VS Code usa esos scopes para saber “qué tipo de cosa” está viendo, y el tema activo (Dark+, Monokai, etc.) contiene reglas que asignan un color a cada scope. En otras palabras: la gramática decide la *clase* del fragmento; el tema decide el *color*.

Si el objetivo es algo más avanzado (por ejemplo, colorear distinto un identificador dependiendo de si es clase, función, parámetro o constante), entonces el enfoque por expresiones regulares se queda corto. Ahí entra el semantic highlighting: en vez de basarse solo en patrones de texto, se apoya en análisis real del lenguaje (lexer + parser + tabla de símbolos). En la práctica esto se implementa mediante semantic tokens provistos por una extensión o, más típicamente, por un Language Server (LSP). VS Code recibe esos tokens semánticos y el tema define la coloración con reglas separadas para categorías semánticas. La idea importante es esta: si solo quieres “colorear por forma” (regex), TextMate es suficiente; si se quiere “colorear por significado”, se necesita un análisis semántico y un proveedor de tokens semánticos.

8. Investigue otra herramienta similar a Flex y de una breve descripción de cómo funciona. Puede que más adelante tenga que realizar actividades con dicha herramienta.

Una herramienta muy utilizada como alternativa moderna a Flex es ANTLR (ANother Tool for Language Recognition). A diferencia de Flex, que se centra principalmente en el análisis léxico (generación de un lexer), ANTLR está diseñado para generar automáticamente tanto el lexer como el parser a partir de una gramática. Esto lo vuelve especialmente útil cuando el objetivo no es solo reconocer tokens, sino también

comprender la estructura del lenguaje (por ejemplo, expresiones, sentencias, bloques, precedencia de operadores, etc.). En ANTLR se escribe una gramática en un archivo `.g4` donde se definen reglas léxicas (tokens) y reglas sintácticas (producciones del parser). Con esa gramática, ANTLR produce código en varios lenguajes (Java, C#, JavaScript, Go, entre otros) que implementa el lexer y el parser.

El flujo típico consiste en generar el código, compilarlo y luego ejecutar el parser sobre un archivo de entrada. El resultado no es únicamente una secuencia de tokens, sino normalmente un árbol de parseo (parse tree) que se puede recorrer con patrones como *visitor* o *listener* para ejecutar acciones: validar, interpretar, traducir, construir un AST, o implementar fases posteriores de un compilador. En resumen, ANTLR se parece a Flex en la parte léxica, pero da un paso adicional es que permite pasar de “reconocer piezas” a “reconocer estructura”, lo cual facilita implementar lenguajes o analizadores más completos y mantenibles cuando el problema crece.

Video de funcionalidad: https://www.youtube.com/watch?v=LAXMMGFULKU