

Morales Victor Eduardo

Sesiones del Analizador Léxico

Fecha

03-10-24

analizador léxico.

• Pasa al scanner un token.
• Los caracteres de entrada
• mas.

• A cada lexema
• esa tabla en símbols.
• tabla para identificar el
• para análisis sintáctico.

Unidad

parser → to semantic analysis

#3

• Usos de espacioado.
• Es de nuevo línea.
• Cero de línea en fuente,
• errar en fuente.
• También expanden las macros.

Analizador Léxico

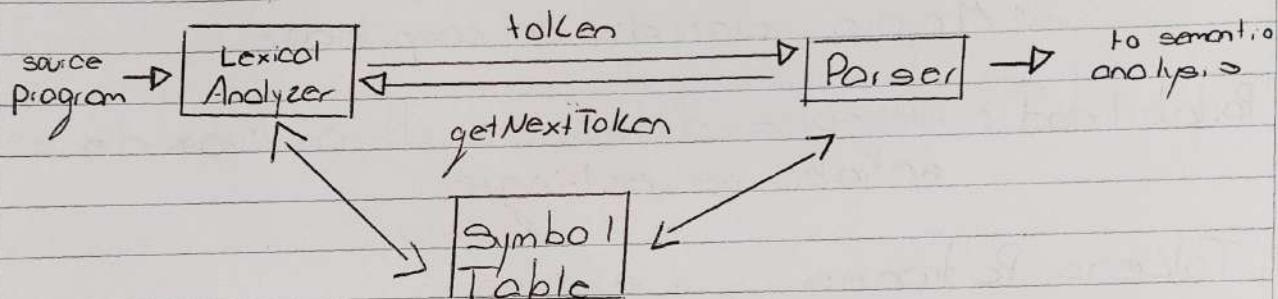
Léxico puede dividirse en 2 procesos:

- Elimina comentarios
- Compacta espacios en blanco consecutivos.

El rol del analizador léxico.

U3

- El parser solicita al scanner un token
- El scanner lee los caracteres de entrada
- Conforma los lexemas.
- Produce tokens para cada lexema
- Inserta lexema con esa tabla en símbols.
- Interactúa con esa tabla para identificar el token correcto.
- Devuelve tokens al parser para análisis sintáctico.



Otras tareas del analizador léxico:

- Eliminar comentarios y caracteres de espacioado.
- Mantener conteo de caracteres de nuevo línea.
- Asociar errores con número de línea en fuente.
- Insertar mensajes de error en fuente.
- Algunos compiladores también expanden las macros.

Procesos del analizador léxico

El analizador léxico puede dividirse en 2 procesos:

- 1) Escanea:
 - Elimina comentarios
 - Compacta espacios en blanco consecutivos

- 2) Análisis Léxico :
- Proceso más complejo
 - Genera tokens del resultado del escaneo

Razones para Separar Análisis Léxico y Sintáctico

- Simplicidad :
- Facilita diseño de compiladores
 - Parser más simple sin espacio y comentarios

- Eficiencia :
- Técnicas especializadas mejoran análisis léxico.
 - Mejora velocidad del compilador

- Portabilidad :
- Peculiaridades de dispositivos de entrada se restringen

Tokens, Patrones y Lexemas

Token : Por nombre -valor, representa un díl exico, ej: palabra clave.

Patrón : Describe la forma del lexema; Para palabra clave, es la secuencia de caracteres.

Lexema : Secuencia de caracteres que coincide con el patrón, identificado como una instancia de un token.

Token

if	caracteres i, f	if
else	caracteres e, l, s, e	else
comparación	< or > or <= or >= or = = or !=	<=, !=
id	letras seguidas de ^{1 o más} digitos	p., score, D2
número	cualquier constante numérica	3.14159, 0, 6.023
literal	cualquier cosa entre " " sin los " " corredorped	

```
printf("Total = %. d\n", score);
```

Tanto printf como score son lexemas que coinciden con el patrón de token id y Total = %. d \n" es un ejemplo que coincide con literal.

Atributos con Tokens

- Varios lexemas pueden coincidir con un patrón
- El analizador léxico, brinda información adicional. Ejemplo: Token "número" puede ser 0 a 1
- Se devuelve nombre del token y su valor atributo
- El nombre afecta el miosis; el atributo lo traducción
- Tokens tienen un atributo, que puede contener varios datos.
- Caso "id", atributo apuntará a de tabla correspondiente.
- Ciertos tokens operadores, puntuación, palabras clave no requieren atributos.
- Un token "número" tendrá un atributo comprobatorio.
- Realmente, el compilador almacena la constante como cadena.
- El valor atributo para "número" es un puntero a esa cadena.

$$E = M * C ** 2$$

se escribirá como una secuencia de pares.

<id, puntero a entrada en la tabla de símbolos para E>

<assign_op>

<id, puntero a entrada en la tabla de símbolos M>

<mult_op>

<id, puntero a otro entrada en la tabla para símbolos C>

<exp_op>

<número, valor entero 2>

Errores léxicos

- Difícil para analizador léxico detectar errores de código.
- El ; " ; " puede ser un final de ";" o un identificador no declarado.
- El analizador léxico devuelve token "id" al parsear.
- El parser o fases posteriores manejan el error.
- Si no encuentra token válido, procede a "modo pónico", que es eliminar caracteres hasta encontrar token válido.
- Puede confundir al parsear, pero es adecuado en entornos interactivos.

Otros estrategias de recuperación de errores

- Eliminar un carácter del input constante.
- Insertar un carácter faltante.
- Reemplazar un carácter por otro.
- Transponer 2 caracteres adyacentes.
- Transformar el prefijo para obtener lexema válido.

Gonzalez Morales Victor Eduardo

Fecha
03/10/20

- Mayoría de errores léxicos son de un solo carácter.
- Correciones más complejas implicativas debido a alto costo.

Divide los sig. programas

03/10/24

C++:

```
float limiteSquare (x) float x; {  
    // Retorna x cuadrado, pero no más de 100 *  
    // return (x <= -10.0 || x > 10.0)? 100: x * x;  
}
```

Java:

```
public static double CalcularAreaCirculo(double  
radio) { double area = Math.PI *  
Math.pow(radio, 2);  
return area;
```

}

Python:

x = 5

if x > 9:

```
print ("! Hola Mundo! ")
```

¿Qué lexemas deben tener palabras léxicas
adequadas? ¿Cuáles debieran ser esas palabras?

TOKEN	Descripción informal	Lexemas múltiples	Lexemas en el programa	Palabras léxicas adecuadas
; f	caracteres ;, f	;	f	NA
else	caracteres e, l, s, o	else	NA	NA
comparación	<=, >=, ==, !=, <, >	<, !=	NA	NA

Gonzalez Moroles Victor Eduardo

Fecha
03/10ct

```
int num1 = 5;
int num2 = 10;
```

```
public static int sumar (int a, int b) {
    int suma = a + b;
    System.out.println ("La suma es: " + suma);
}
```

TOKEN	Descrip informal	Lexemas muestra	Lexemas en el programa	valores lógicos asociados
public	caracteres p,u,b,l,i,c	public	public	NA
static	caracteres s,t,a,t,i,c	static	static	NA
assign- op	=, + =, - =, *=, /=, % =	=, / =	=	=
add- op	*	+	+	NA
id	Letras seguidas de letra y digitos	num1, num2, num3,	num 1	puntero a entrada en la tabla de simbolos para num1
número	Cualquier constante numérica	3.14159, 8	5	5
literal	Cualquier cosa entre los ""	una cadena de palabras	La suma es	puntero a entrada en la tabla de simbolos para la suma es.
curly left	{	{	{	NA
puntuación	, ; , :	:	:	;

Gonzalez Morales Victor Eduardo

Fecha
03/Oct

En el caso del token assign-op, en el ejemplo, se da como descripción informal el conjunto de operadores de asignación definidos en los diversos lenguajes de programación existentes, es por eso que después es necesario decir cuál fue el que se leyó del código, en la columna de valores léxicos asociados, lo mismo sucede para el token "puntuación"

En el caso del token add-op como nadamas es un símbolo puede ser add-op, pues no se necesita agregarle atributo, por eso, dice NA en la columna de los valores léxicos asociados, lo mismo sucede con el token cuty left.

Python $x = 5$

; f $x > 9$

print ("Hola mundo")

07/10ct

TOKEN	DESC Informal	Lexemas muestra	Lexemas programa	Valores
id		X	X	
assign-op	=, +, ==, +=, /=, *=	=, /s	=	=
num	cualquier constante		5	5
; f	cualquier carácter	; f	; f	NA
puntuación	<, >, == !=, <, >	< = >	< = >	
num	cualquier constante	5, 9	9	9
Puntuación	;	;	;	

TOKEN	DESCRIPCION	LEXEMA	LEXEMAS PROGRAMA	VALORES
Print	carácteres	print	print	N/A
par 129	print, + (((N/A
LITERAL	cualquier cosa ente " " o "	"coredump"	Hola mundo	Puntero a entrada en tabla de símbolos para hola mundo.
PARA)))	N/A

09 Oct +

Manejo de Buffers de Entrada

Buffer de entrada.

Se necesita mirar más allá del próximo lexema.

Ej: No se sabe si un identificador termina hasta ver un carácter no alfanumérico.

Operadores en C pueden ser de uno o dos caracteres.

Se usa un esquema de dos buffers para manejar el look ahead.

Los buffers permiten anticipar varios caracteres con seguridad.

La mejora "sentinelas" reduce el tiempo en checar el fin de los buffers.

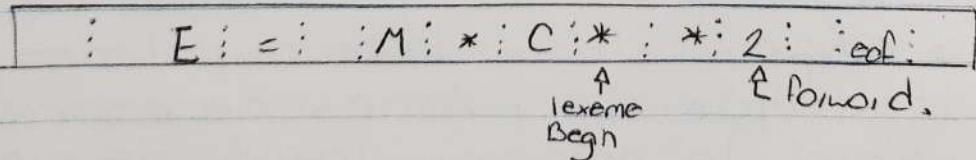
El buffer acelera la lectura del programa fuente.

Técnica de Pares de Buffers

- Justificación: Procesar eficientemente grandes volúmenes de caracteres.
- Dos buffers altanos se reaigan para evitar sobre cargo.
- Cada buffer tiene tamaño N , 4096 bytes.
- Un comando de lectura cargo N caracteres en lugar de uno por uno.
- Si quedan menos de N caracteres, eof marca el final del archivo.
- El puntero lexema Begin marca el inicio del lexema.
- El puntero forward avanza hasta encontrar una coincidencia de patrón.

Estrategia de Punteros en pares de buffers

- Al encontrar un lexema, forward se mueve a lo derecho del último.
- Lexema Begin se ajusta al carácter después del lexema.
- El puntero forward puede regresar si se pasa del lexema.



Estrategias de Punteros

- Se aprueba si forward alcanza el final de un buffer.
- Si es así, se recarga el otro buffer y el forward se mueve al inicio.
- Mientras no se sobrepase el tamaño N , el lexema no se sobrescribe.

Gonzalez Morales Victor, Eduardo

Fecha: 09/10ct

- Garantiza la detección del lexema antes de perderlo en el búfer.

Sentinelas en Búferes

- Se debe verificar si forward sale de búfer.
- Si sale, se recarga el otro búfer.
- Se realizan 2 pruebas por cada carácter leído.
- El uso de sentinelas simplifica esta verificación.
- Es un carácter especial, como eof.
- eof marca el fin del búfer o del archivo completo.
- El algoritmo solo prueba el carácter actual, salvo al final de un búfer.

```
| : |E| = | : |M| *eof| C| *| *| 2 |eof| ;eof|  
↑ ↑  
lexeme forward  
Begin.
```

Especificación de los Componentes Léxicos

11/10ct

Especificación de tokens:

- expresión - regular, especifican patrones de lexemas.
- Efectivos aunque no cubren todos los patrones posibles.
- Definen los tipos de patrones necesarios.
- Se estudiará la notación formal de expresiones regulares.
- Se analizará su uso en generadores de analizadores léxicos.
- Se explicará la construcción de expresiones regulares en automatos.
- Automatos reconocen tokens específicos.

Cadenas y Alfabetos

- alfabeto conjunto finito de símbolos
- Ejemplos comunes: letras, dígitos, puntuación.
- $\{0, 1\}$ alfabeto binario, ASCII, Unicode.
- cadena secuencia finita de símbolos de un alfabeto.
- Longitud de una cadena se denota $|s|$.
- Cadena vacía, denotada ϵ , tiene longitud cero.

Lenguajes y Concatenación

- lenguajes conjunto contable de cadenas sobre un alfabeto.
- Lenguajes abstractos incluyen \emptyset (conjunto vacío) y $\{\epsilon\}$.
- Ejemplos: programas C bien formados, frases gramaticales en español.
- concatenación de 2 cadenas x y y es xy .
- Ej: si $x = \text{"pasu"}$ y $y = \text{"mecho"}$; entonces $xy = \text{"pasumecho"}$.
- La cadena vacía es la identidad en la concatenación: $\epsilon s = s \epsilon = s$.
- "exponentiación" de cadenas: $s^2 = ss, s^3 = sss$, etc.

Términos para partes de cadenas

- Prefijo: Elimina símbolos del final de la cadena (ej: ban, banana, ϵ).
- Sufijo: Elimina símbolos del inicio (ej: nana, banana, ϵ).
- Subcadena: Elimina cualquier prefijo y sufijo (ej: nan, banana, ϵ).

- Prefijo / Sufijo / Subcadena propia: Igual que el prefijo / sufijo / subcadena pero no es ni la cadena completa.
- Subsecuencia: Formada al eliminar cero o más símbolos no consecutivos (ej: ban de banana)

Operaciones con Lenguajes:

- Unión: Combina los elementos de 2 lenguajes.
- Concatenación: Une una cadena de un lenguaje con una del otro.
- Clasura Kleene (L^*): Concatena el lenguaje L cero o más veces.
- L^0 : Es $\{\epsilon\}$, la concatenación de L cero veces.
- Clasura Positiva (L^+): Concatena L al menos una vez.
- L^+ no incluye $\{\epsilon\}$ a menos que L lo contenga.

Operación

Unión de L y M

Concatenación de L y M

Cerradura de Kleene de L

Cerradura positiva de L

Definición y notación

$L \cup M = \{s | s \text{ está en } L \text{ o } s \text{ está en } M\}$

$LM = \{st | s \text{ está en } L \text{ y } t \text{ en } M\}$

$L^* = \bigcup_{i=0}^{\infty} L^i$

$L^+ = \bigcup_{i=1}^{\infty} L^i$

Expresiones Regulares

- Describen lenguajes usando operadores: Unión, concatenación y clausura.

- Ej: Identificadores en un lenguaje.

- 'letter' representa letras o guioncitos bajos,
- 'dig+' representa dígitos

- Un identificador válido se escribe como:
 - 'letter - (letter | digit)*'
- ' '|=union, '*' = cero o más ocurrencias
- Se constituyen recursivamente usando subexpresiones
- Cada expresión denota un lenguaje.

Reglas básicas de expresiones regulares

Das reglas fundamentales.

- 1) 'ε' es ER, $L(\epsilon) = \{\epsilon\}$, el lenguaje con solo la cadena vacía.
- 2) Si 'a' es un símbolo en Σ, entonces 'a' es uno ER y $L(a) = \{a\}$, un lenguaje con una cadena de longitud 1.

Construcción por inducción de expresiones regulares

- Se construyen expresiones más grandes a partir de otras más pequeñas.

- 1) '(r)' | '(s)' denota $L(r) \cup L(s)$
- 2) '(r)(s)' denota $L(r)L(s)$
- 3) '(r)*' denota $L(r)^*$.
- 4) '(r)' denota $L(r)$ (agregar paréntesis no cambia el lenguaje).

Simplificación de expresiones regulares.

- Expresiones regulares a menudo contienen paréntesis innecesarios.
- Podemos eliminarlos si seguimos las convenciones:

- 1) El operador '*' tiene la mayor precedencia y es asociativo a la izquierda.
- 2) La concatenación tiene la 2da precedencia más alta y es asociativo a la izquierda.
- 3) '**' tiene la menor precedencia y es asociativo a la izquierda.

• Ej: '(a)1((b)* (c))' se puede simplificar a 'a1b*c'.

Conjuntos regulares y leyes algebraicas.

- Un lenguaje definido por una expresión regular se llama en conjunto regular.
- Si 2 expresiones regulares denotan el mismo conjunto regular, son equivalentes.
- Ej: '(a1b)= (b1a)'.
- Las leyes algebraicas afirman que expresiones diferentes pueden ser equivalentes.
- Estas leyes se aplican a expresiones regulares 'r', 's', y 't' arbitrarias.

LEY

DESCRIPCIÓN

$$r1s = s1r$$

1 es commutativo

$$r1(s1t) = (r1s)1t$$

1 es asociativo

$$r(s+t) = (r s)+$$

La concatenación es asocitativa

$$r(s1t) = r s1 t; (s1t)r = s1t r$$

11 se distribuye a favor de 1

$$\epsilon_r = r \epsilon = r$$

ϵ es la identidad para la concatenación

$$r* = (r1\epsilon)^*$$

ϵ es garantizada en un cierre

$$r** = r^*$$

* es idempotente.

Definiciones regulares

- Para simplificar asignar nombres a ER.
- Definición regular secuencia de la forma:
 - $d_1 \rightarrow r_1$
 - $d_2 \rightarrow r_2$
- Cada ' d_i ' es un símbolo nuevo, cada ' r_i ' es una ER.
- Los ' r_i ' utilizan el alfabeto Σ y las definiciones previas.
- Se evitan definiciones recursivas.
- Se reemplazan los ' d_i ' gradualmente para obtener expresiones regulares completas.

Extensiones de Expresiones Regulares.

1) Una o más instancias (+):

- Representación clausura positiva.
- $(r)^+$ denota $L(r)^+$.

2) Cero o uno instancia (?) :

- $r^?$ es equivalente a $r^* \cup \epsilon$.

3) Clases de caracteres:

- $a \mid b \mid c$ puede escribirse como $[a, bc]$.
- Rango lógico, como $[a - z]$, representa $a \mid b \mid \dots \mid z$.

Especificación de las Componentes Léxicos
ejemplos

Operaciones con Lenguajes - Ejemplo

Sea: L el conjunto de letras $\{A; B; \dots; z; a; b; \dots; z\}$

D el conjunto de dígitos $\{0; 1 \dots; 9\}$.

Podemos pensar en L y D de 2 maneras, esencialmente equivalentes. Una forma es que L y D son los alfabetos de letas y mayus, minus y de dígitos. La 2da forma es que L y D son lenguajes, cuyas cadenas tienen todas longitud uno. Aquí hay algunas otras lenguajes que se pueden construir a partir de los idénticos L y D , usando los operadores vistos.

$$1) LUD$$

$$2) LD$$

$$3) L^4$$

$$4) L^*$$

$$5) L(LUD)^*$$

$$6) D^+$$

1) LUD es el conjunto de datos de letras y dígitos, es hablando, lenguaje con 62 cadenas de longitud 1, cada una de las cuales es una letra o un dígito.

2) LD es el conjunto de 520 cadenas de longitud dos, cada una de las cuales consta de una letra seguida de un dígito.

- 3) L^4 es el conjunto de todas las cadenas de 4 letras.
- 4) L^* es el conjunto de todas las cadenas de letras, incluida la cadena vacía.
- 5) $L(LUD)^*$ es el conjunto de todas las cadenas de letras y dígitos que comienzan con una letra.
- 6) D^* es el conjunto de todas las cadenas de uno a más dígitos.

Ej: Sea $\Sigma = \{a; b\}$

1. $a|b$
2. $(a|b)(a|b)$
3. a^*
4. $(a|b)^*$
5. $a|a^*b$

1. $a|b$ denota el lenguaje $\{a, b\}$.
2. $(a|b)(a|b)$ denota $\{aa, ab, ba, bb\}$, el lenguaje de todas las cadenas de longitud dos sobre el alfabeto Σ . Otra expresión regular para el mismo lenguaje es $aabb|baabbb$.
3. a^* denota el lenguaje que consta de todas las cadenas de cero o más a , es decir, $\{\epsilon, a, aa, aaa, \dots\}$.
4. $(a|b)^*$ denota el conjunto de todas las cadenas de cero o más instancias de a o b , es decir, todas las cadenas de a y b : $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$.
5. $a|a^*b$ denota el lenguaje $\{a; b, ab, aab, \dots\}$, es decir, la cadena a y todos los cadenas que constan de cero o más a y terminan en b .

Gonzalez Morales Victor Eduardo

Fecha

11 Oct

Definiciones Regulares - Ejemplo.

Ej # 1.- Los identificadores de Java son cadenas de letras, dígitos y guiones bajos. Estructura una definición regular para el lenguaje de los identificadores de Java.

letra $\rightarrow A \mid B \mid \dots \mid z \mid a \mid b \mid \dots \mid z \mid -$
dígito $\rightarrow 0 \mid 1 \mid \dots \mid 9$
id $\rightarrow \text{letra} (\text{letra} \mid \text{dígito})^*$

Ej # 2 - Los números sin signo (entero o punto flotante) son cadenas como 5280, 0.01234, 6, 336E4, o 1.89E-4. Genera una definición regular.

dígito $\rightarrow 0 \mid 1 \mid \dots \mid 9$
dígitos $\rightarrow \text{dígito dígito}^*$
fracción Opcional $\rightarrow . \text{dígitos} \mid \epsilon$
exponente Opcional $\rightarrow (E \mid + \mid -) \text{dígitos} \mid \epsilon$
número $\rightarrow \text{dígitos fracción Opcional exponente Opcional}$

Extensiones de Expresiones Regulares - Ejemplo

Ej : Usando extensiones de EB, reescribe la definición regular para los identificadores de Java:

letra $\rightarrow [A - Z \mid a - z \mid _]$
dígito $\rightarrow [0 - 9]$
id $\rightarrow \text{letra} (\text{letra} \mid \text{dígito})^*$

González Morales Víctor Eduardo

Fecha
11 Oct

Ej: Usando extensiones de EER, reescribo la definición regular para los números enteros sin signo:

dígito $\rightarrow [0-9]$

dígitos \rightarrow dígito⁺
número \rightarrow dígitos (dígitos)? (E [+ -])? dígitos)?

Reconocimiento de

15 Oct

ToLens

- Expresamos patrones con expresiones regulares.
- Objetivo: construir código que identifica lexemas en la cadena de entrada.
- ToLens a reconocer: 'if', 'then', 'else', 'relop', 'id', 'numero'.
- Comparaciones: como en Pascal o SQL: '=' es igual, '<' es diferente.
- Los patrones de 'id' y 'numero' son EER's.
- Palabras clave serán reservadas, no identificables.

stmt \rightarrow if expr then stmt

| if expr then stmt else stmt

| ε

expr \rightarrow term relop term

| term

term \rightarrow id

| number

González Molinos Víctor, Eduardo

Fecha
15/Oct

digit → [0-9]
digits → digit +
number → digits(.digits)? (E [+ -]?, digits)?
letter → [A-zA-Z]
id → letter (letter | digit)*
if → if
then → then
else → else
relop → < | > | <= | >= | = | <>

Eliminar espacios en Blanco

- Responsabilidad del analizador léxico, eliminar blancos: ws...
- Blancos: Espacios, tabulaciones y nuevas líneas.
- El token ws no se devuelve al analizador.
- Análisis léxico se reinicia tras espacio blanco.
- Sólo el siguiente token es devuelto al analizador.

ws → (blank | tab | newline) *

Objetivo del Analizador Léxico.

- Asignar tokens a cada lexema o familia de lexemas
- Devolver el nombre del token y valor de atributos a parser.
- Para operadores relacionales, se usan constantes simbólicas: 'LT', 'LE', etc.
- Constantes necesarias para indicar instrucciones encontrada de relop.
- La tabla resume esta información para cada token reconocido.

LEXEMES	TOKEN NAME	ATTRIBUTE	VALUE
Any ws	-	-	-
if	;f	-	-
then	then	-	-
else	else	-	-
Any id	:d	Pointer to table entry	
Any number	number	Pointer to table entry	
<	relOp	LT	
<=	relOp	LE	
=	relOp	EQ	
>	relOp	NE	
>=	relOp	GT	
>=	relOp	GE	

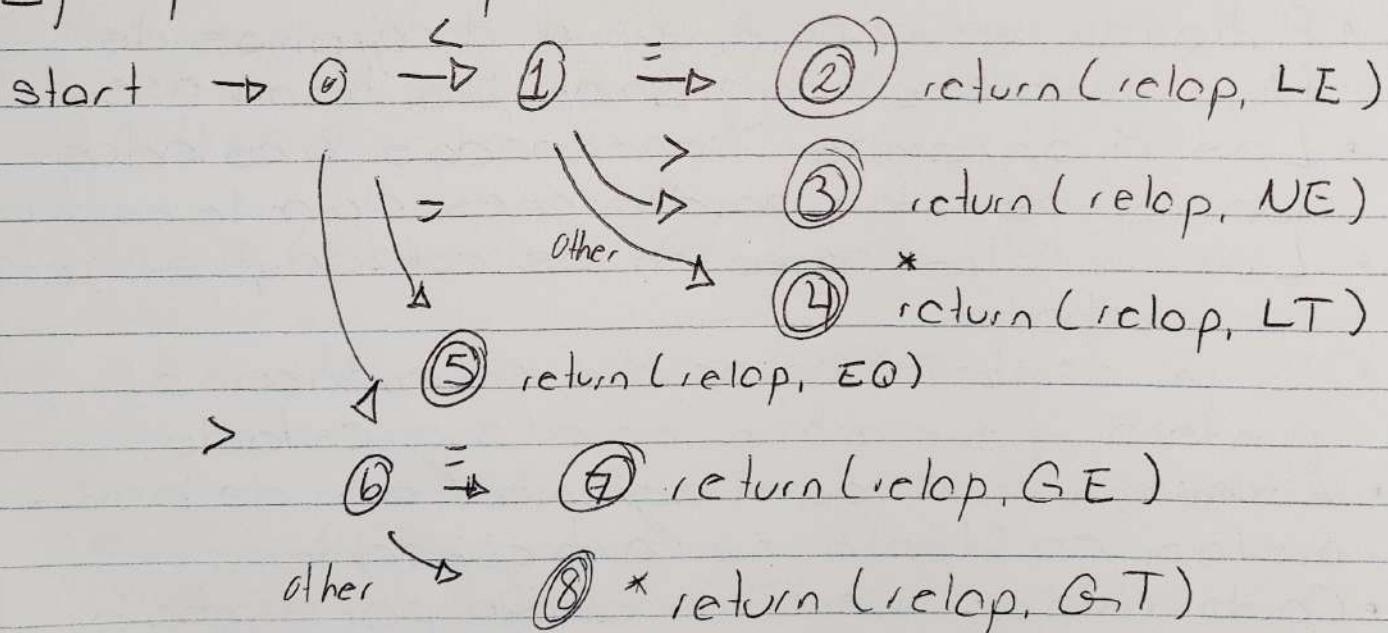
Diagramas de Transición - Concepto.

- Páginas se convierten en diagramas de flujo llamadas diagramas de transición
- Los diagramas tienen nodos o estados, que representan condiciones durante escaneo.
- Los aristas conectan los estados, etiquetados con símbolos
- Si un símbolo coincide, se avanza el puntero y se entra en el sig estado.
- Asumimos que los diagramas son deterministas, es decir, sin ambigüedades.
- Cada estado tiene un arista por símbolo o conjunto de símbolos
- Al procesar un símbolo, el puntero forward avanza y el estado cambia.
- Se simplifica la búsqueda de lexemas durante el análisis léxico:
 - Asegurando una única transición posible por símbolo (determinismo).
 - La condición de determinismo se lleva a cabo en etapas avanzadas del diseño léxico.

Convenciones de los Diagramas de Transición.

- Estados de aceptación indican que se encontró un lexema
- Se representan con un doble círculo y pueden reformar un token
- Si se debe retroceder el puntero forward, se coloca un *.
- Estados iniciales comienzan el proceso, indicados con un orilla "Inicio"
- El diagrama siempre empieza en el estado inicial antes de tener símbolos.
- En este ejemplo, nunca se retrocede más de una posición.

Ejemplo Relop



Reconocimiento de Palabras Clave e Identificadores

- Problemo: Los palabras clave como : for then parecen identificadores.

- Las palabras clave están reservadas, no son identificadores
 - El mismo diagrama reconoce palabras clave e identificadores
- start → ⑨ → ⑩ → ⑪ * return getToken()
letter dig. → other → install D()

Manejo de Palabras Reservadas - Método 1

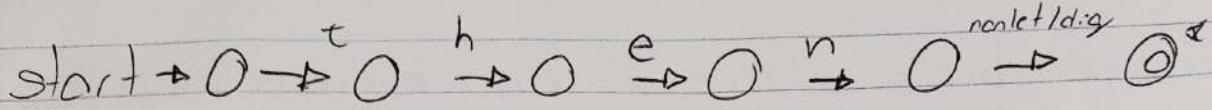
- Instalar palabras reservadas en la tabla de símbolos.
- Un campo en la tabla indica que no son identificadores.
- Al encontrar un identificador, instalo la coloco en la tabla.
- Si el identificador no está en la tabla, su token es id.
- getToken revisa la tabla y devuelve el nombre del Token.
- Tokens pueden ser id o palabras clave prioritario
- Método usado en el ejemplo.

Método 2

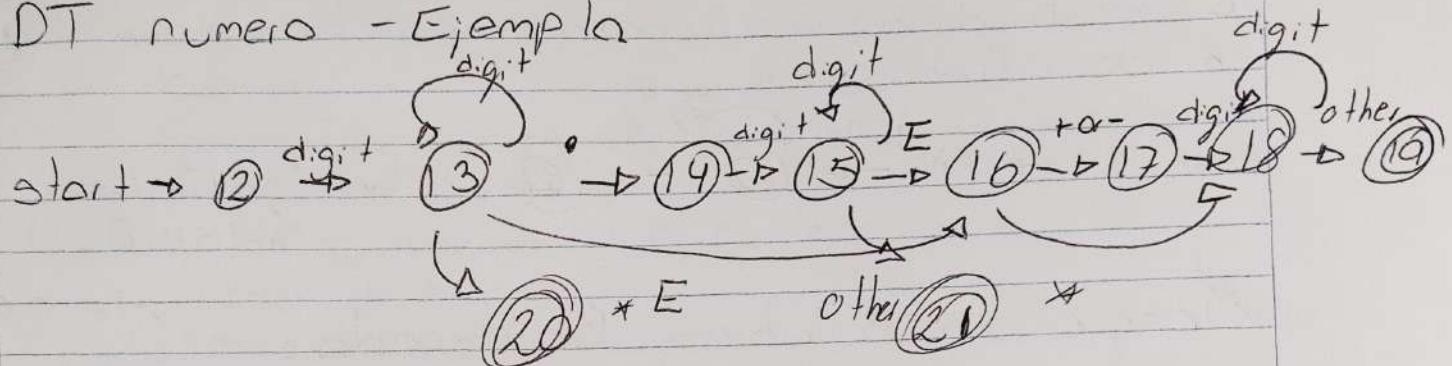
- Crear diagramas de transición separados para cada palabra clave.
- Ejemplo: diagrama de la palabra clave then.
- Verifica que el identificador haya finalizado.
- Evita confundir lexemas como thenvalue con then.
- Priorizar palabras reservadas sobre id.
- No se usó este enfoque en nuestro ejemplo.

Gonzalez Maroles Victor Eduardo

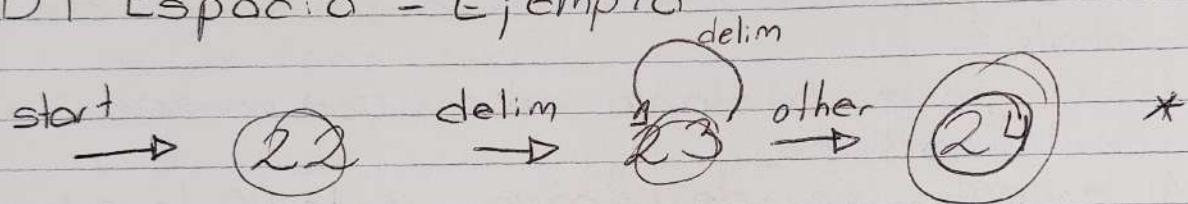
Fecha 17 Oct +



DT numero - Ejemplo



DT Espacio - Ejemplo



Arquitectura de un
Analizador Léxico

17 Oct +

- DT's producen analizador léxico
- Cada estado tiene un fragmento de código.
- La variable static almacena el estado actual.
- Un switch se usa para determinar las acciones por estado.
- El código cambia el siguiente carácter de entrada.
- Cambio de estado según el carácter leído.
- Implementa bifurcación o instrucciones múltiples.

getRelop()

{ Token getRelop()

```
TOKEN retToken = new(IRELOP);
while (1) {
    switch(state) {
        case 0: c = nextChar();
            if (c == '<') state = 1;
            else if (c == '=') state = 5;
            else if (c == '>') state = 6;
            else fail();
            break;
        case 1: ...
```

```
        case 8: retract();
            retToken.attribute = GT;
            return (retToken);
```

3 Simulación de Diagramas de Transición

- Organizar diagramas por token
- Usar funciones 'fail()' para reiniciar puntero.
- Palabras clave como reservados.
- Ejecutar diagramas "en paralelo".
- Resolver coincidencias de patrones.
- Preferir coincidencias de prefijo largo.
- Combinar diagramas en uno solo.

Gonzalez Morales Victor Eduardo

17 Oct

TOKEN getRelOp {

TOKEN retToken = new TOKEN();

while (1) {

switch (state) {

case 0: c = nextChar();

if (c == '<') state = 1;

else if (c == '=') state = 5;

else if (c == '>') state = 6;

else fail();

break;

case 1: c = nextChar();

if (c == '=') state = 2;

else if (c == '>') state = 3;

else state = ;

break;

case 2: retToken.setToken("RELOP");

retToken.setCode("LE")

return (retToken); break;

Simulación de Diagramas
de Transición - Aplicación.

21 Oct

Para lenguaje generado por la expresión regular ($a^1 b^2 + a^3 b^2$, crear el DTC (grafo)
generar la tabla de transiciones y
un programa para implementar el analiza-
dar léxico correspondiente a la tabb
que imprima "cadena aceptada" o
"cadena no aceptada" según corresponda.

Gonzalez Morales Victor Eduardo

Fecha: 21/10/0+

Cadena ababba bobba ababb - Resultado:
CADENA NO ACEPTADA

... Program finished with exit code 0
Press ENTER to exit console.

Cadena ababba babb - Resultado: CADENA
ACEPTADA

... Program finished with exit code 0
Press ENTER to exit console.

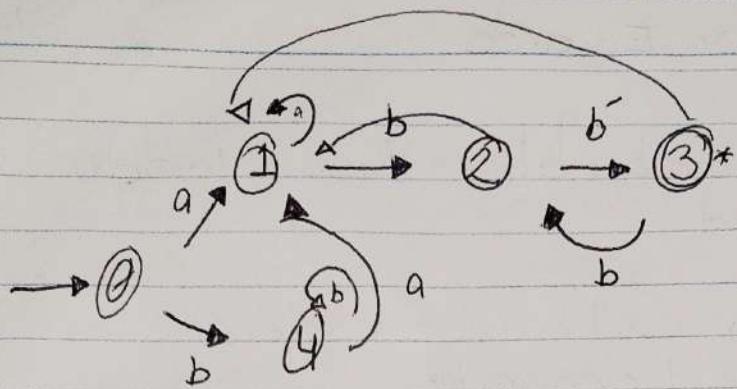
case 3 retToken.setToken('Relop');
retToken.setCode('NE');
break;

case 4 retract();
retToken.setAttribute = LT;
return (retToken);
break;

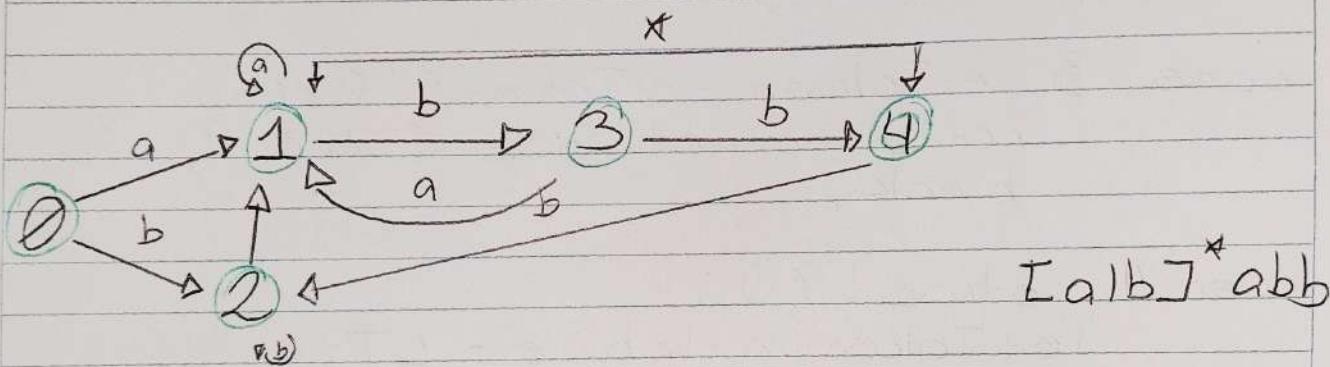
case 5 : retToken.setToken("REMP");
retToken.setCode("EQ");
return (retToken);
break;

case 6 : C = nextChar();
, if (C == '=') state = 7;
else state = 8;
break;

case 7 : retToken.setToken('Relop');
retToken.setCode('GE');
break;



- $\alpha^1 b^2 b^3$ Acepto abbabbabb
- $a^1 b^2 a^1$ NA
- $a^1 b^2 a^1 b^2 b^3$ A
1 2 3 3 3



STATE	a	b	Tolken	Retroceso
0	1	2	-	-
1	1	3	-	-
2	1	2	-	-
3	1	4	-	-
4	1	2	Aceptado	-

- 1) inicio
- 2) Pedir una cadena para analizar y guardar la cadena en la variable "cadena"
- 3) Repetir lo siguiente desde el 1er carácter hasta el último carácter de "cadena"
 - Leer siguiente carácter y guardarlo en la variable "dato"

1: Inicio:

2-Inicializa variable "estado" en cero

3- Pedir una cadena para analizar y guardar la cadena en la variable "cadena"

4- Repetir lo siguiente desde el primer caracter hasta el ultimo caracter "cadena"

4.1 En caso de que "estado" sea:

cero: leer siguiente caracter de "cadena" y guardarlo en "dato"

si "dato" es uno 'a' entonces guardar en "estado" un uno.

si no si "dato" es uno 'b' entonces guardar en "estado" un dos.

si no invocar a rutina de error.

fin caso cero.

una: leer siguiente caracter de "cadena" y guardar en "dato".

si "dato" es una 'b' entonces guardar en "estado" un tres.

sino si "dato" no es una 'a' entonces invocar a rutina de error.

fin caso una.

dos: leer siguiente caracter de "cadena" y guardar en "dato".

si "dato" es una 'a' entonces guardar en "estado" en ver uno.

sino si "dato" es una 'b' entonces invocar una rutina de error

fin caso dos.

González Moroles Víctor Eduardo

Fecha 24/10/ct

Tres: leer siguiente carácter de "cadena" y guarda en "dato"

Si "dato" es una 'o' guardar en "estado" uno

Sino si "dato" es una 'b' entonces

guardar en "estado" cuatro

Sino invocar a rutina de error

Fin caso tres.

Cuatro: leer siguiente carácter de "cadena" y guardarlo en "dato"

Si "dato" es una 'o' entonces guardar en "estado" uno

Si "dato" es una 'b' entonces guardar en "estado" dos

Sino sino hay más caracteres imprimir

"CADENA ACEPTADA"

Sino invocar a rutina de error

Fin caso cuatro.

Fin de casos.

5- Si "estado" no es 4 entonces

Imprimir "CADENA RECHAZADA"

6- Fin.

1- Inicio

2- Pedir una cadena y analizar y guardarla en la variable "cadena"

3- Repetir lo sig desde el 1er carácter hasta el último carácter de "cadena"

Ley. sig carácter y guardarla en la variable "dato"

Si encontramos "o", empazamos a buscar una "b"

Si después de "a" encontramos una b, buscamos otro "b".

Si encontramos la 2da "b" la cadena es válida.

Si en cualquier momento encontramos algo diferente, volvemos a empezar.

4- Si terminamos encontrando "abb", mostrar "cadena aceptada". Si no mostrar "cadena rechazada".

5- Fin

	a	b
0	1	2
1	1	3
2	1	2
3	1	4
4	1	2 Acepta

Generadores de analizadores léxicos 28 Oct

Herramientas para separar cadenas leídas en elementos lexicográficos que corresponden a tokens.

Lex → C

Flex → C

P Cllex

ANTLR → Java, C, C++, C#, Python, Perl, Delphi, Ada 95, Javascript, Objective-C.

González Morales Víctor Eduardo

Fecha 28-Oct

CUP → Java

Cocktail - Rex

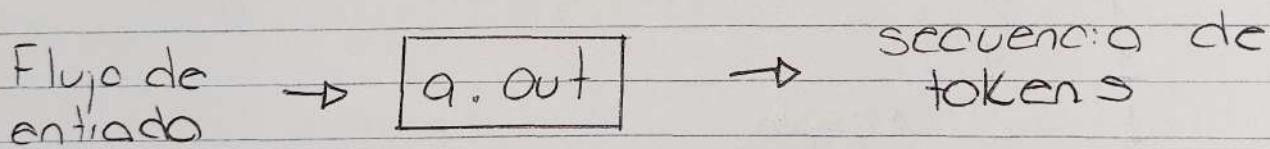
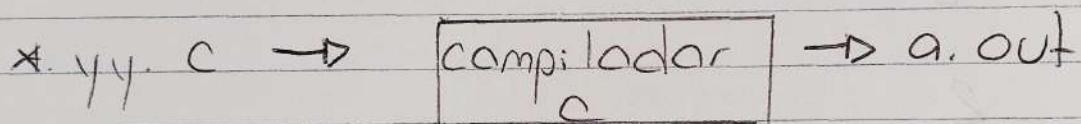
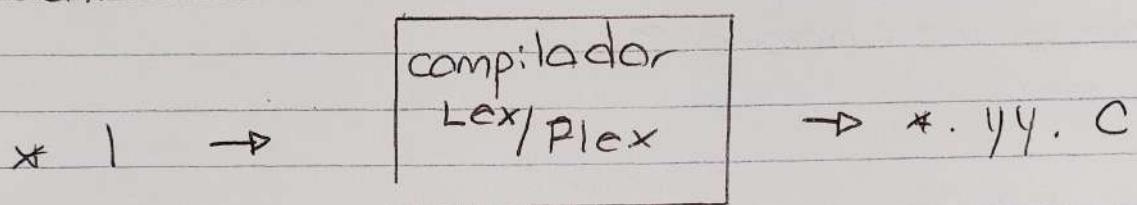
PCCT - DLG

Generan 'lexicos' automáticamente

Enfoque: Lex, Flex

Archivero

Fuente



Estructura de una fuente para Lex

declaraciones => declaración de variables
% %.

reglas => pares patrón - acción
% %.

funciones auxiliares => funciones que pueden compilarse aparte

Ej: %?

#include <stdio.h>

int valores = 0, consonantes = 0;

Máster

3%

Gonzalez Morales Victor Eduardo

Fecha

%. %.
[aeiouAEIOU] { vocales ++; }
(bcdflghjklmnprstuvwxyzBCDFGHJKLMNPQRSTUVWXYZ) { consonantes ++; }
. \n;
. %.
%

29 Oct

Ingrese una cadena: Esto es:

Una linea mas
otra linea. Para ver si funciona:
vocales minusculas: 17
consonantes minusculas: 15
vocales mayúsculas: 4
consonantes mayúsculas: 6
Otras simbolos : 1
No. de lineas: 3

int yywarp() { #2
 return 1;
}

```
int main() {  
    printf("Ingrese una cadena: ");  
    yylex();  
    printf("Vocales minúsculas: %d\n", vocales);  
    printf("Consonantes minúsculas: %d\n", consonantes);  
    printf("Vocales mayúsculas: %d\n", vocalesmayus);  
    printf("Consonantes mayúsculas: %d\n", consonantesmayus);  
    printf("Otras simbolos: %d\n", caracter);  
    printf("No. de lineas: %d\n", saltoFinal);  
    return 0;
```

Emraloz Morales Victor Eduardo

Fecha

29 Oct

1

```
% {  
#include <stdio.h>  
int vocales = 0, consonantes =  
0, vocalesMayus = 0, consonantesMayus = 0,  
saltoLinea = 0, caracter = 0;  
% }  
% /*  
[aeiou] { ++vocales; }  
[AEIOU] { vocalesMayus; }  
[bedfghijklmnpqrstuwyz] { ++consonantes; }  
[BCDFGHIJKLMNOPQRSTUVWXYZ] {  
    ++consonantesMayus; }  
[\n] { ++saltoLinea; }  
[.] { ++caracter; }  
*/
```

Espec. Finales
Léxicas.

30 Oct

Un archivo de especificación léxica para JFlex
consta de 3 partes divididas por una sola
línea que comienza con /* */.

User Code

/* */

Opciones y declaraciones

/* */

Reglas Léxicas

En todas las partes de la especificación,
se permiten comentarios de la forma
/* */ o texto de comentario /* */ que
de final de línea de estilo Java que
comienzan con /*. Los comentarios de JFlex
se anidan, por lo que el número de /*

Rayter

Gonzalez Morales Victor Eduardo

Fecha
30/10/20+

y λ^* debe estar equilibrado.

Ej: Cómo trabajar con JFlex?

Para demostrar cómo se ve una especificación léxica con JFlex, esta sección presenta un ejemplo que cuenta vocales y consonantes en una cadena ingresada por teclado:

Archivo LetiasLexer.flex:

% %

% class Letiaslexer

% unicode

% standalone

% {

private int vocales = 0;

private int consonantes = 0;

private int lineas = 0;

public void printCounts() {

System.out.println("Número de vocales:
" + vocales);

System.out.println("Número de consonantes:
" + consonantes);

System.out.println("Número de lineas:" +
lineas);

}

% }

VOCAL = [aeiouAEIOUáéíóúÁÉÍÓÚ]

CONSONANTE = [q-zA-Z&[^aeiouAEIOUáéíóúÁÉÍÓÚ
ÁÉÍÓÚ]]

NEW LINE = [\r|\r\n]

1. 1.

{VOCAL} {vocales ++;}
{consonante} {consonantes ++;}
{NEW-LINE} {lincas ++;}

Código a Incluir

La primera sección, el código de usuario:
El texto hasta la primera linea que comienza con i. i. se copia literalmente en la parte superior de la clase de lexergenerado (antes de la declaración de clase `reol`). Junto con las declaraciones de package e import.
Por lo general no hay mucho que hacer.

Sí el código termina con un comentario de clase javadoc, la clase generada obtendrá este comentario, sino, JFlex generará uno automáticamente.

Opciones y declaraciones (macros)

La segunda sección, opciones y declaraciones, es más interesante. Consiste en un conjunto de opciones, código que se incluye dentro de la clase de escáner generada, estás lexicas y declaraciones de macros. Cada opción JFlex debe componer una linea de especificación y comenzar con un i. En nuestro ejemplo se utilizan las sig opciones:

- 1. class LetrasLexer, le dice a JFlex que le dé a la clase generada el nombre LetrasLexer y que escriba el código en un archivo LetrasLexer.java.
- 1. unioade define el conjunto de caracteres en los que trabajará el escáner.
- 1. standalone indica que el analizador léxico generado será un programa independiente. Esto significa que el analizador no dependerá de otro programa o entorno para funcionar.

El código entre 1. E y 13 se copia literal en el código fuente de la clase lexer generada. Aquí puede declarar variables miembro y funciones que se utilizan dentro de las acciones del escáner. Al igual que con todas las acciones de JFlex, tanto 1. { como 1. ; deben comenzar una línea.

La especificación comienza con declaraciones de macros. Las macros son abreviaturas de especificaciones regulares, que se utilizan para facilitar la lectura y comprensión de las especificaciones léxicas. Una declaración de macro consta de un identificador de macro seguido de = y de la expresión regular que representa. Esta expresión regular, puede contener o su vez usos de macros.

En el ejemplo se definen las variables vocales, consonantes y líneas para almacenar

los conteos de vocales, constantes y líneas así como el método printCount que imprime los valores, consonantes y líneas de cada contador.

Aunque esto permite un estilo de especificación similar a la gramática, las macros siguen siendo solo abreviaturas y no terminales, no pueden ser recursivas. JFlex detecta y notifica las ciclas en las definiciones de macros en el momento de la generación.

Reglas y acciones

La sección contiene expresión regulares y acciones que se ejecutan cuando el escáner coincide con la expresión regular asociada. A medida que el escáner lee su entrada, realiza un seguimiento de todas las expresiones regulares y activa la acción de la expresión que tiene lo largo. Si 2 expresiones regulares tienen la coincidencia más larga para una entrada determinada, el analizador elige la acción de la expresión que aparece primero en la especificación.

Aquí cada expresión regular tiene una acción que incrementa el contador correspondiente:

- Si se encuentra una vocal, se incrementa vocales,
- Si se encuentra una consonante, incrementa consonantes.
- Si se encuentra un salto de línea, incrementa líneas.
- Otros caracteres (puntos, espacios, etc.) se ignoran.

Gonzalez Morales Victor Eduardo

Fecha
31/0ct

Ademas de los coincidencias de expresiones regulares, se pueden utilizar estados léxicos para refinar. Un estado léxico oculta como una condición de inicio.

Implementación y ejecución.

Para implementar el analizador léxico generado con JFlex en Java y contar vocales, consonantes y líneas desde consola, necesitarás 2 archivos principales:

- 1- El archivo .flex con las especificaciones para JFlex
- 2- El archivo .java principal que ejecuta el analizador.

Para generar el analizador léxico en Java a partir del archivo .flex, ejecuta el sig. comando en la terminal:

jflex LetrasLexer.flex

Esto generará un archivo llamado LetrasLexer.java en el mismo directorio. Aquí se muestra el código de InputLexer.java para crear el archivo java que se iniciará y ejecutará después leerá una cadena de texto desde el teclado y utilizará el analizador para procesarla.

Se usa un java.io.String Reader para permitir que JFlex lea una cadena proporcionada por el usuario directamente desde el teclado.

```
import java.io.StringReader;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;

public class InputLexer {
    public static void main (String [] args) {
        try {
            BufferedReader reader = new BufferedReader (new InputStreamReader (System.in));
            System.out.print ("Ingresar lo que deseo a analizar:");
            String inputText = reader.readLine();
            LetrasLexer lexer = new LetrasLexer (new StringReader (inputText));
            int token;
            while ((token = lexer.yylex ()) != LetrasLexer.EOF) {
                lexer.printCounts ();
            }
            catch (IOException e) {
                System.out.println ("Error al leer la entrada:");
                + e.getMessage ();
            }
        }
    }
}
```

González Morales Víctor Eduardo

Fecha
31-0ct

Explicación:

Se utiliza BufferedReader junto con InputStreamReader(System.in) para leer lo introducido desde el teclado.

Se utiliza StringReader para crear una instancia del analizador léxico LetrasLexer y pasarsela la cadena leída.

El resto del código procesa la entrada carácter a carácter, llamando a lexer.yy.lex() y luego se imprime el conteo de vocales, consonantes y líneas.

Para ejecutar el analizador, primero se compilan ambas archivos .java en un terminal:

java o InputLexer, java LetrasLexer, java

Luego, ejecuto el programa java InputLexer

Ej. de ejecución:

Ingreso la cadena a analizar:

Hola, cómo estás?

Número de vocales: 6

Número de consonantes: 7

Número de líneas: 1

Gonzalez Morales Victor Eduardo

Fecha

4/11/2018

Hacer un lexer stand alone completo en JFlex
y la clase que lo invoca para que leo
desde un archivo de texto plano las
cadenas de texto en cada linea y reco-
nozco como tokens: números enteros, números
hexadecimales, números octales e identi-
ficadores, usando los siguientes:

Letra: Cualquier letra (mayúscula o minúscula).

Dígito: Cualquier dígito (0 a 9)

Alfanumérico: cualquier letra o dígito.

Número: secuencia de dígitos (0 o más).

Entero: un entero positivo (comenzando en 0
o en un dígito distinto de 0)

Octal: un número en base 8, comenzando con 0

Hexadecimal: un número hexadecimal, comen-
zando con 0x.

Identificador: un identificador que comienza
con una letra y puede tener letras, dígitos
y un guión bajo (`_`) después de la
primera letra.

Cuando identifiques algún lexema que cumple
con algún patrón debe escribirse en un archivo
de texto plano el lexema rellenando
el token correspondiente a ese lexema
y un salto de linea.

Letra = [a - zA - Z]

Dígito = [0 - 9]

Alfanumérico: {LETRA} | {Dígitos}

Número: {Dígitos} +

Gonzalez Morales Victor Eduardo

Fecha
05-nov

Entero = 0 | {Numeros}
Octal 0 [0-7] +

HEXADECIMAL = 0x | x[0-9 a-f A-F] +
IDENTIFICADOR = Letras - ?
& ALFANUMÉRICOS +

01

0103

02

0102

102 ← Entero

0x0-9

0x

07 - NOV

```
import java.io.*;  
public class Main {  
    public static void main(String[] args) {  
        if (args.length < 2) {  
            System.out.println("Uso : java Lexer  
Main <archivo-entrada><archivo-salida>");  
            return;  
        }  
        String archivoEntrada = args[0];  
        String archivoSalida = args[1];  
        try {  
            BufferedReader reader = new BufferedReader(  
                new FileReader(archivoEntrada));  
            BufferedWriter writer = new BufferedWriter(  
                new FileWriter(archivoSalida));  
        }  
        LexerClass lexer = new LexerClass  
        (reader, writer);  
        while (true) {
```

```
lexer.yylex();  
if(lexer.isEOF()) break;  
}
```

```
System.out.println("Análisis completado.  
Tokens guardados en " + archivoSalida);  
} catch (IOException e) {  
e.printStackTrace();  
}  
}
```

```
-----  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.io.Reader;
```

```
/*  
 * public  
 * class LexerClass  
 * extends  
 * standalone  
 */
```

```
private BufferedWriter writer;  
public boolean isEOF() {  
return zzA+EOF; }  
}
```

```
public LexerClass(Reader in, BufferedWriter  
writer) {  
this.zzReader = in;  
this.writer = writer;  
}
```

```
private void escribeToken(String lexema,  
String token) {  
try {
```

González Marañón Víctor Eduardo

Fecha: 07-nov

```
writer.write(cxema + " - " + token + "\n");  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

1.3
3

DIGITO = [0 - 9]

LETRA = [a - zA - Z]

ALFANUMERICO = {LETRA} {DIGITO}*
IDENTIFICADOR = {LETRA} {ALFANUMERICO}

NUMERO = {DIGITO}*

N-LINEA = "\n"

OTRO = [. \d] +

1. 1.

< yyinitial > {

{ IDENTIFICADOR } EscribeToken(yyText(),
 "IDENTIFICADOR"). 3

{ numero } EscribeToken(yyText(), "numero"); 3

{ N-LINEA } { /* Ignorar cualquier salto */ 13 }

{ OTRO } { /* Ignorar otro carácter */ 3 }

3

ols 1 12 vc - Identificador

0564 - Número

98547 - Número

mbj - Identificador

zib - Identificador

33 - Número

s - Identificador

0 - Número

XABZ3 - Identificador

105 - Número

F - Identificador

11-nov

Fecha

Agregar al lexer en proceso la siguientes palabras reservadas, operadores simbolos
capacidad para reconocer las
int - INT
float - FLOAT
char - CHAR
CharString - CHARSTRING
if - IF
else - ELSE
do - DO
repeat - REPEAT
for - FOR
begin block -
BEGINBLOCK
end bloc K -
ENDBLOCK
while -
WHILE
(- LEFTPART
) - RIGHTPART
read - READ
end program

< - LT RELATION
> - GT RELATION
<= - EQ RELATION
>= - EG RELATION
:= - EQ RELATION
~= - NEQ RELATION
+ - SUMOP
- - SUBOP
* - PRODOP
/ - QUOTOP
= - ASIGNAOP
and - ANDLOG
or - ORLOG
not - NEGLOG
display - DISPLAY
read - READ

Contenido del Archivo de
Lectura: in.txt
program MI_PROG;
begin block
int x,y;
float pocketo;
display ("DAME X: ~");
read(x);
display ("DAME Y: ~");
read(y);
pocketo = x * y;
if (pocketo > 100)
begin block
display ("CORRECTO");
end block
end program

Mayter

González Morales Víctor Eduardo

Fecha
12/11/2017

IF = "if"

FLOAT = "float"

ELSE = "else"

CHAR = "char"

PARENTESIS = "("

DO = "do"

INT = "int"

REPEAT = "repeat"

DIGITO = [0 - 9]

LETRA = [a - z A - Z]

ALFANUMERICO = {LETRA} {DIGITOS}

IDENTIFICADOR = {LETRA} {ALFANUMERICOS}

NUMERO = {DIGITOS} +

N-LINEA = "\n"

OTRO = [. \r \t]

;

yy init();

{ IDENTIFICADOR } { escribirToken(yy init());
IDENTIFICADOR "); }

{ NUMEROS } { escribirToken(yy text(), "NUMERO"); }

{ N-LINEAS } { /* Ignorar saltos de linea */ }

{ OTROS } { /* Ignorar otro caracter */ }

{ PARENTESIS } { escribirToken(yy text(), "PARENTESIS"); }

{ INT } { escribirToken(yy text(), "INT"); }

{ FLOAT } { escribirToken(yy text(), "FLOAT"); }

{ CHAR } { escribirToken(yy text(), "CHAR"); }

{ DO } { escribirToken(yy text(), "DO"); }

{ REPEAT } { escribirToken(yy text(), "REPEAT"); }