Informe

Analizador léxico

Mauro Montano LU:108882

Analizador léxico

La implementación del analizador léxico realizo haciendo uso del lenguaje de programación JAVA. Dicho analizador convierte una entrada de caracteres en una entrada de Tokens, donde cada Token se corresponde con un lexema de MINIJAVA, además, cada Token tiene asociado un número de línea de código en la cual el mismo fue encontrado.

Compilar y ejecutar

El programa fue desarrollado en Eclipse y para realizar una compilación desde línea de comandos, es necesario situarse en la carpeta donde tenemos el exportable .jar y ejecutar el siguiente comando:

java –jar Etapa1.jar <Entrada> [Salida]

Donde Entrada es un parámetro obligatorio, con la ruta del archivo fuente a ser evaluado en el análisis léxico incluyendo el formato de archivo. Por otro lado, Salida es un parámetro opcional que representa el archivo de texto donde se guardará el resultado del análisis léxico junto con su formato, o en caso de que el archivo de Salida no sea especificado, el resultado se mostrará por pantalla.

Alfabeto de entrada

El analizador Léxico reconoce y permite cualquier caracter unicode. Los caracteres especiales como por ejemplo @ se permiten en el programa pero no forman parte de tokens válidos. Es decir, pueden aparecer en comentarios o en literales caracteres o strings, pero en cualquier otra parte del programa se detectarán como caracteres inválidos.

Tokens

TOKEN	EXPRESION REGULAR
IdMetVar	(a..z)(A..Z + a...z + 0..9 + _)*
IdClase	(A..Z)(A..Z + a...z + 0..9 + _)*
Lexema entero	[0-9]+[0-9]*
Lexema_String	\"*+\"
Lexema_Caracter	\'+[^\s|^\\]{1}\' | \'\\+[^\s]{1}\'
EOF	EOF
PR_Extends	extends
PR_Class	class
PR_Void	void
PR_Boolean	boolean
PR_If	if
PR_Else	else
PR_This	this
PR_New	new
PR_Static	static
PR_Dynamic	dynamic
PR_Public	public
PR_Char	char
PR_Int	int
PR_String	String
PR_While	while
PR_Return	return
PR_Null	null
PR_True	true
PR_False	false
PR_Protected	protected
PR_Final	final
P_Parentesis_Abre	\(

P_Parentesis_Cierra	\)
P_Llave_Abre	{
P_Llave_Cierra	}
P_Coma	,
P_PuntoComa	;
P_Punto	\.
O_Menor	<
O_Mayor	>
O_Not	!
O_Comparacion	==
O_Asignacion	=
O_Distinto	¡=
O_Menorigual	<=
O_Mayorigual	>=
O_Resta	-
O_Suma	\+
O_Mult	*
O_Div	/
O_Mod	%
O_And	&&
O_Or	||

Errores léxicos detectados

Errores generales del programa

La clase Principal define un conjunto de errores generales ajenos al analizador. Estos errores son:

“Se esperaban argumentos”: Surge cuando no se introduce un archivo a analizar:

<INPUT_FILE>

“Demasiados argumentos”: Aparecerá cuando se introducen 3 o más archivos.

“El archivo de entrada es inválido”: Si surge algún problema con los archivos de entrada.

Errores detectados

El analizador léxico es capaz de reconocer los errores léxicos que se enumeran a continuación:

CharMalFormado: Caracteres mal formados, por ejemplo cuando se intenta incluir más de un caracter entre las comillas simples (quotes) que delimitan un caracter. Formato: " Error lexico en linea '"+nroLinea+"': "+chr+" no es parte de un caracter bien formado"

CharInvalido: Caracteres que no pertenecen al alfabeto. Formato: " Error lexico en linea '"+nroLinea+"': "+chr+" no es un caracter valido"

IdentificadorMalFormado: El valor de un literal entero esta mal formado, por ejemplo: 1515Mauro. Formato: " Error lexico en linea '"+nroLinea+"': "+chr+" no es parte de un identificador bien formado"

StringMalFormado: Cadenas (String) mal formadas, por ejemplo, cuando se incluye un salto de línea antes del cierre de la cadena. Formato: " Error lexico en linea '"+nroLinea+"': "+chr+" no es parte de un string bien formado"

ComentarioMalFormado: Comentarios mal formados, como por ejemplo, un comentario multilinea sin cerrar. Formato: " Error lexico en linea '"+nroLinea+"': "+chr+" no es parte de un comentario bien formado"

OperadorInválido: Se informa el error cuando el usuario ha ingresado solamente un simbolo “&” o una barra vertical “|” para expresar un AND o un OR, pero se ha olvidado del caracter que le sigue a continuación. Formato: " Error lexico en linea '"+nroLinea+"': "+chr+" no es un operador valido"

La ejecución del analizador termina inmediatamente cuando un error es encontrado.

Si no hay ningún error se informa con el siguiente mensaje: "Análisis léxico operó de forma exitosa"

Clases implementadas

Estructuras

Token

Estructura de datos simple para guardar un Token. El token posee un nombre (que representa el tipo), un lexema, el número de línea y el numero de columna. Los métodos de la estructura son:

- **getNombre()**: devuelve el nombre (tipo) del Token.

- **getLexema()**: devuelve el lexema propiamente dicho.
- **getLinea()**: devuelve el número de línea donde fue encontrado el token.
- **getColumna()**: devuelve el número de columna donde fue encontrado el token

TablaTokens

Esta clase fue realizada con un HashMap que representa una tabla con las palabras reservadas y el tipo de token. Luego, se agregaron los otros tipos de tokens a la tabla (símbolos de puntuación, operadores, etc.). Esta posee dos métodos:

- **esPalabraReservada(cadena)**: retorna *true* si una cadena de texto es una palabra reservada (también puede ser usada para los símbolos del lenguaje).
- **obtenerTipo(cadena)**: retorna el tipo de token para una determinada cadena. Si la cadena no fuera un token reconocido, devuelve *null*.

Helper

Provee un conjunto de métodos para simplificar la lectura y la verificación de algunas estructuras condicionales en el analizador léxico. Posee siguientes métodos:

- **esLetra(letra)**: retorna *true* si un caracter dado es una letra (a-z ó A-Z).
- **esMayuscula(letra)**: retorna *true* si un caracter dado es una mayúscula (A-Z).
- **esDigito(letra)**: retorna *true* si un caracter dado es un dígito (0-9).
- **esGuionBajo(letra)**: retorna *true* si un caracter dado es un guión bajo (_).
- **esSeparador(letra)**: retorna *true* si un caracter dado es un separador. Esto es, un espacio, un enter, o una tabulación.
- **esEnter(letra)**: retorna true si un caracter dado es un *Enter*.
- **esIdentificador(letra)**: retorna *true* si un caracter dado es una letra, un dígito o un underscore. Cualquiera de las tres opciones forman parte de posibles caracteres válidos dentro de un identificador escrito correctamente.
- **esEOF(letra)**: retorna *true* si un caracter dado representa un *End of File*.
- **esComillas(letra)**: retorna *true* si un caracter dado es comillas dobles.
- **esApostrofo(letra)**: retorna *true* si un caracter dado es una comilla simple.

Módulos

Main

Se encarga de la lógica de archivos del programa (y la interacción con el usuario). Su función principal es crear un analizador léxico y pedirle tokens en base al archivo de entrada que haya provisto el usuario.

AnalizadorLexico

Se función es hacer el análisis léxico de acuerdo al *buffer de lectura* que se le haya asignado. Posee el único método público:

- **getToken**(): Busca según su posición actual en el *buffer*, formando y retornando un objeto de tipo *Token* si así fuera posible. En caso de no ser posible, en analizador lanzará una excepción informando el error encontrado.

Excepciones

Posee implementadas las clases de los errores léxicos especificados anteriormente en la sección de Errores Léxicos Detectados.

Decisiones de diseño

- El módulo Principal muestra los tokens en un formato de tabla. Los tokens cuyo lexema tenga una longitud mayor a 32 caracteres (por ejemplo, un identificador o un string) pueden llegar a salirse del formato de la tabla sin embargo no es recurrente que ocurra y tampoco se considera de importancia para ésta etapa del proyecto. Lo mismo ocurre con el número de línea el cual tiene una longitud no mayor a 8 dígitos.

- La especificación con respecto a los comentarios provista por la cátedra define a los comentarios simples como iniciados por un // y finalizados en un Enter. Para este caso, el analizador léxico implementado también reconoce los comentarios que están al final del archivo y que terminan con un EOF.

- Al detectar errores léxicos, como por ejemplo un error de String mal formado que falten cerrar las ultimas comillas: “mal . En casos como este, se marca el error en el lugar siguiente al la letra “l” , por lo tanto, se marcará el error en la columna 5 y se dirá que un espacio vacío no es parte de un string bien formado ya que debería haber una comilla “ en ese lugar.

- Los lexemas correspondientes a los caracteres \n y \t se guardan como strings, para permitir reflejar el lexema en la tabla mostrada.
- La clase *BufferedReader* provista por *Java* no guarda a *End of File* como un caracter, sino que en caso de que no queden elementos en el *buffer* para leer, retornará un -1. Teniendo en cuenta esto, se considera que cuando se encuentre con el caracter que *casteado* da -1, será un *End of File*.

Casos de prueba

Test validos

- **Test1** (Palabras reservadas): Se verifica que el analizador sea capaz de identificar las palabras reservadas del lenguaje. Primero una por línea, y luego todas en la misma línea.
- **Test2** (Identificadores): Se escribieron identificadores válidos, algunos de clase y otros de variable (o métodos). Debería mostrarlos todos y llegar al token de *End of File*.
- **Test3** (Comentarios): Para verificar distintos usos de los comentarios, tanto en línea simple como multilínea. Algunos comentarios poseen caracteres pertenecientes al alfabeto pero no a identificadores de tokens. El análisis debería devolver únicamente el token *End of File*.
- **Test4** (Enteros): Se prueba que se reconozcan los enteros .
- **Test5** (Strings): Se prueba reconocer correctamente los strings.
- **Test6** (EOF): El archivo vacío debería devolver únicamente el token de *End of File*.
- **Test7** (Operadores): Se prueba el uso de operadores aritméticos y lógicos, en líneas separadas, así como también en la misma línea separados por un espacio, o todos consecutivos y se le agregó algunos *Enters* al principio del archivo para variar.
- **Test8**(Caracteres): Se prueba que el analizador reconozca correctamente distintos tipos de literales caracteres. Se prueba literales caracteres que estén separados, consecutivos, haciendo uso de la barra invertida (\), caracteres especiales como nueva línea o tabulación, operadores y otros caracteres del alfabeto.
- **Test9** (Puntuación): Se prueban los símbolos de puntuación, en líneas separadas, así como también en la misma línea separados por un espacio, o todos consecutivos.
- **Test10** (General): Un programa de tokens están separados por espacios intencionalmente.

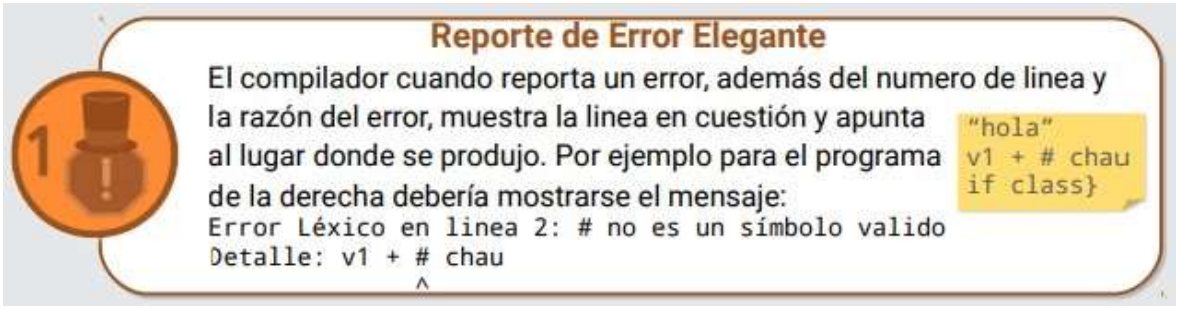
Test Invalidos

- **Test1 (Identificadores):** El analizador debe parar y reportar el error ya que el identificador comienza con un "_".
- **Test2 (Identificadores):** El analizador debe reportar un carácter inválido siendo éste un "@" mezclado en el identificador.
- **Test3 (Identificador mal formado):** Muestro que un identificador mal formado como: 1515mauro es un error.
- **Test4 (Clase con Operador AND mal formado):** Se espera un reporte de error por caracter tener un operador mal formado dentro de un "*if*" donde encontramos un solo "&".
- **Test5 (Operador OR mal formado):** Se espera un reporte de error por operador OR mal formado "|".
- **Test6 (Carácter mal formado):** Se espera otro reporte de error por literal caracter mal formado. En este caso, el error se debe al exceso de caracteres posibles en un literal caracter.
- **Test7 (Carácter mal formado):** Se espera un reporte de error por literal caracter mal formado '\' , donde se esperaría un apóstrofo más al final.
- **Test8 (Carácter mal formado):** Se espera otro reporte de error por literal caracter mal formado. En este caso se omitió poner algún otro caracter entre las comillas simples.
- **Tes9 (Carácter mal formado):** Se espera otro reporte de error por literal caracter mal formado. En este caso se ingresan más de un carácter numérico, y además se olvida de cerrar comillas simples aunque éste último error no llega a detectarlo por encontrar otro antes.
- **Test10 (Comentario multilínea mal formado):** Un comentario multilínea que no es correctamente cerrado.
- **Test11 (Comentario multilínea mal formado):** Otro comentario multilínea mal formado, en donde se usa el mismo asterisco (*) de la apertura para su cierre (que debería ser erróneo).
- **Test12(Operador AND mal formado):** Se espera un reporte de error por operador AND mal formado "&".

- **Test13 (String mal formado):** Un literal string que abre comillas dobles pero no son cerradas (se llegará a *End of File* antes de encontrar las siguientes comillas dobles)
- **Test14 (String mal formado multilínea):** Un literal string que ocupa mas de una linea no está contemplado por MiniJava. Por lo tanto, se notifica el error de StringMalFormado.

Logros

Reporte de error elegante

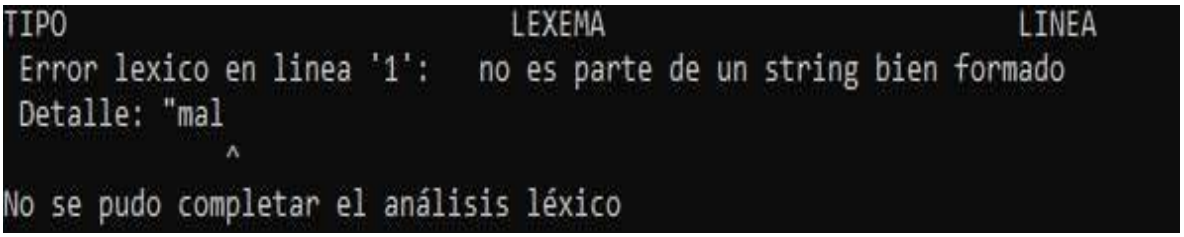


El logro se implementó como se especifica aquí arriba, usando el símbolo exponente " ^ " para apuntar al error en la línea de Detalle. Para implementarlo, hice métodos privados como consumirLinea() que una vez que se encuentra un error consume el resto de la línea y la guarda, para luego mostrarla en la parte de Detalle, luego fui contando las columnas para guardar la columna donde encuentro el error y por último, para apuntar al error lo hago en cada clase Excepcion determinada.

Algunos ejemplos:

```
TIPO                        LEXEMA                      LINEA
PR_idMetVar                 este                        1
 Error lexico en linea '1': @ no es un caracter valido
 Detalle: este@esuncaracterinvalido
               ^
No se pudo completar el análisis léxico
```

Aquí arriba se puede ver el ejemplo de la línea este@esuncaracterinvalido donde se explica que el arroba no es un carácter valido y luego en detalle se muestra la línea y se apunta al error determinado.



Como mencioné en las decisiones de diseño, en el caso de un string mal formado, si tengo “mal estaría faltando las comillas que cierran, por lo tanto, se apuntará en el detalle que donde deberían estar las comillas estas no se encuentran y luego más arriba en la línea de error léxico en línea ‘1’ se pondrá un espacio en blanco.