Proyecto Compiladores e Interpretes

Etapa 1

Analizador Léxico

Rodrigo Díaz Figueroa

LU 97400

Clases utilizadas y decisiones de diseño

Clases:

AnalizadorLexico
Principal
EntradaSalida
Token
Utilidades
LexicoException

Principal:
Es la clase principal que tiene el método main.
Esta clase se encarga de crear objetos de las clases EntradaSalida y AnalizadorLexico, asociarlos y utilizar los métodos de estas clases para imprimir los tokens por pantalla o en el archivo opcional.

AnalizadorLexico:
Se encarga del análisis léxico del archivo de entrada, devolver objetos de tipo Token con el método nextToken() y lanzar excepciones de tipo LexicoException cuando hubo un error léxico.
Utiliza la clase EntradaSalida para obtener los caracteres del archivo.

EntradaSalida:
En general se encarga de abrir, cerrar y crear los archivos de entrada y salida.
Se encarga de leer carácter por carácter el archivo de entrada utilizando las clases FileReader y BufferedReader provistas por Java, y de crear y escribir lineas en el archivo opcional de salida, para esto utiliza las clases FileWriter y BufferedWriter provistas por Java.
Esta es la clase encargada de contar las lineas y las columnas del archivo.

Utilidades:
Esta clase es estática y en ella se encuentran las constantes de tipo entero que representan a los todos tipos de tokens y algunos métodos estáticos que utilizan otras clases.

Token:
Clase que modela un token. Cada token tiene un tipo, lexema, un numero de linea y un numero de columna.
Esta clase también provee un método que utiliza la clase Principal para imprimir el token.

LexicoException:
Clase que hereda de Exception y es utilizada por AnalizadorLexico para lanzar excepciones.

Decisiones de diseño:

- Se utiliza el algoritmo “switch case” para la implementación del autómata reconocedor del analizador léxico.

- El autómata que reconoce los distintos tipos de tokens es reducido, es decir, algunos estados reconocen mas de un tipo de token. En particular el estado inicial (0) reconoce todos los tokens de un solo carácter, el resto de los estados reconocen a lo sumo 2 tipos distintos de tokens.

- Las palabras claves son reconocidas dentro de los estados que reconocen los tokens de tipo Identificador de Método-Variable e Identificador de Clase. Se utiliza la un método que provee la clase Utilidades que retorna el tipo del token según un lexema.

- Los tipos de los tipos de tokens son enteros, se utilizan constantes para facilitar de lectura/escritura.

Observaciones generales:

El programa en ocasiones imprimirá dos tokens de fin de archivo, esto se debe a un bug que todavía no fue corregido pero que no afecta al desempeño del resto del programa. Se cree que este bug se da cuando los archivos fuentes tienen uno o mas caracteres “whitespace” luego del ultimo token y antes del final del archivo.

Logros

Los logros que se intentan alcanzar en esta etapa son:

- **Strings Multilinea:** Reconocer adecuadamente literales Strings con saltos de linea cuando tienen \n dentro

- **Columnas:** Para cada token reconocido también se almacena el numero de columna donde comienza en el archivo fuente

- **Imbatibilidad Léxica:** El software entregado paso correctamente toda la batería de prueba utilizada por la cátedra.

Compilación y Ejecución

Para compilar el programa desde consola se deberá ejecutar el siguiente comando

```
>javac Principal.java
```

Observación: Si este comando no compila todas las clases del programa se deberá ejecutar entonces

```
>javac Principal.java AnalizadorLexico.java Utilidades.java Token.java
EntradaSalida.java LexicoException.java
```

Para la ejecución del programa desde consola se deberá ejecutar

```
>java Principal <archivo fuente> [<archivo salida>]
```

Donde `<archivo fuente>` es el path absoluto o relativo dependiendo de donde se encuentra el archivo fuente a analizar en el equipo.

Y el opcional, `<archivo salida>` es el nombre del archivo de salida donde se escribirá la salida del programa.

Observaciones:

Los errores encontrados serán mostrados por consola incluso si se especifica un archivo de salida.

El programa soporta paths relativos del archivo fuente si se encuentra en el mismo directorio que el resto de los archivos del programa.

El archivo de salida sera creado (o reescrito) en el mismo directorio que se encuentra el programa.

Ejemplos:

```
>java Principal Test/Correctos/Test1.java
```

```
>java Principal Test/Correctos/Test1.java out.txt
```

El archivo Test1.java debe estar dentro del directorio Test/Correctos/ que se encuentra en el mismo directorio que los archivos fuente

```
>java Principal C:/Testing/Incorrectos/Test2.java
```

```
>java Principal C:/Testing/Incorrectos/Test2.java out.txt
```

El archivo Test2.java debe estar dentro del directorio que especifica el path absoluto

Alfabeto

El alfabeto reconocido por el analizador léxico como parte del lenguaje son los símbolos del código ASCII imprimibles. Es decir desde el ASCII n.º 32 ( ) hasta el n.º 126 (~).

***Observación**: En los tokens de tipo Literal Cadena de caracteres existen algunos caracteres que son reconocidos y no son parte del alfabeto (Ej: ¿,â,Â,¥). Esto se debe a que la clase que se utiliza para leer caracteres del archivo de entrada, **BufferedReader**, convierte los caracteres.*

Tokens

Identificador Clase

Token identificador de nombres de clases.

Expresión regular:

IDClase = AX*
A = [A..Z] (letras mayúsculas)
X = _ | [0..9] | [a..z] | [A..Z] (letras mayúsculas, minúsculas, dígitos y guión bajo)

Identificador Metodo-Variable

Token identificador de nombres de métodos o variables.

Expresión regular:

IDMetVar = BX*
B = [a..z] (letras minúsculas)
X = _ | [0..9] | [a..z] | [A..Z] (letras mayúsculas, minúsculas, dígitos y guión bajo)

Literal Entero

Token identificador de literales de tipo enteros.

Expresión regular:

LitEnt = C+
C = [0..9] (dígitos)

Literal String

Token identificador de literales de tipo cadena de caracteres

Expresión regular:

LitString = "Z*"
Z = [...~]
Z es igual a cualquier carácter imprimible del código ASCII, es decir los caracteres desde el n.º 32 ( ) al n.º 126 (~).

***Observación**: El carácter \ (barra invertida) es un carácter de escape dentro de la cadena de caracteres. Este carácter se utiliza para poder ingresar carácter "especiales" dentro de la cadena como un salto de linea (\n), un Tab (\t), la comilla doble (\") y la propia barra invertida (\\).*

*Existen algunos caracteres que no son parte del alfabeto del lenguaje que son admitidos dentro de un token literal cadena de caracteres. Ver **Alfabeto**.*

Literal Caracter

Token que identifica a los literales de tipo caracteres

Expresión regular:

LitCar = 'Z'
Z = [...~]
Z es igual a cualquier carácter imprimible del código ASCII, es decir los caracteres desde el n.º 32 ( ) al n.º 126 (~).

***Observación**: Al igual que con los literales de tipo cadena de caracteres, el carácter \ (barra invertida) es un carácter de escape. Este carácter se utiliza para poder asignar caracteres "especiales" como, salto de linea (\n), Tab (\t), la comilla simple (\') y la propia barra invertida (\\).*

Puntuación punto y coma

Token que identifica a un punto y coma.

Expresión regular:

PunPuntoyComa = ;

Puntuación punto

Token que identifica a un punto.

Expresión regular:

PunPunto = .

Puntuación coma

Token que identifica a una coma.

Expresión regular:

PunComa = ,

Puntuación llave abierta

Token que identifica a una llave que abre

Expresión regular:

PunLlaveA = {

Puntuación llave cerrada

Token que identifica a una llave que cierra

Expresión regular:

PunLlaveC = }

Puntuación paréntesis abierto

Token que identifica a un paréntesis que abre

Expresión regular:

PunParenA = (

Puntuación paréntesis cerrado

Token que identifica a un paréntesis que cierra

Expresión regular:

PunParenC =)

Puntuación corchete abierto

Token que identifica a un corchete que abre

Expresión regular:

PunCorchA = [

Puntuación corchete cerrado

Token que identifica a un corchete que cierra

Expresión regular:

PunCorchC =]

Asignación igual

Token que identifica al operador de asignación

Expresión regular:

AsigIgual = =

Operador igual

Token que identifica al operador de comparación

Expresión regular:

OpIgual = ==

Operador desigual

Token que identifica al operador de desigualdad

Expresión regular:

OpDesigual = !=

Operador negación

Token que identifica a la negación booleana

Expresión regular:

OpNeg = !

Operador mayor

Token que identifica al operador mayor

Expresión regular:

OpMayor = >

Operador mayor igual

Token que identifica al operador mayor igual

Expresión regular:

OpMayorIg = >=

Operador menor

Token que identifica al operador menor

Expresión regular:

OpMenor = <

Operador menor igual

Token que identifica al operador menor igual

Expresión regular:

OpMenorIg = <=

Operador suma

Token que identifica a la operación de suma

Expresión regular:

OpSuma = +

Operador resta

Token que identifica a la operación de resta

Expresión regular:

OpResta = -

Operador multiplicación

Token que identifica a la operación de multiplicación

Expresión regular:

OpMult = *

Operador división

Token que identifica a la operación de división

Expresión regular:

OpDiv = /

Operador AND

Token que identifica a la operación AND

Expresión regular:

OpAND = &&

Operador OR

Token que identifica a la operación OR

Expresión regular:

OpOR = ||

Fin de archivo

Token especial que identifica el fin de archivo

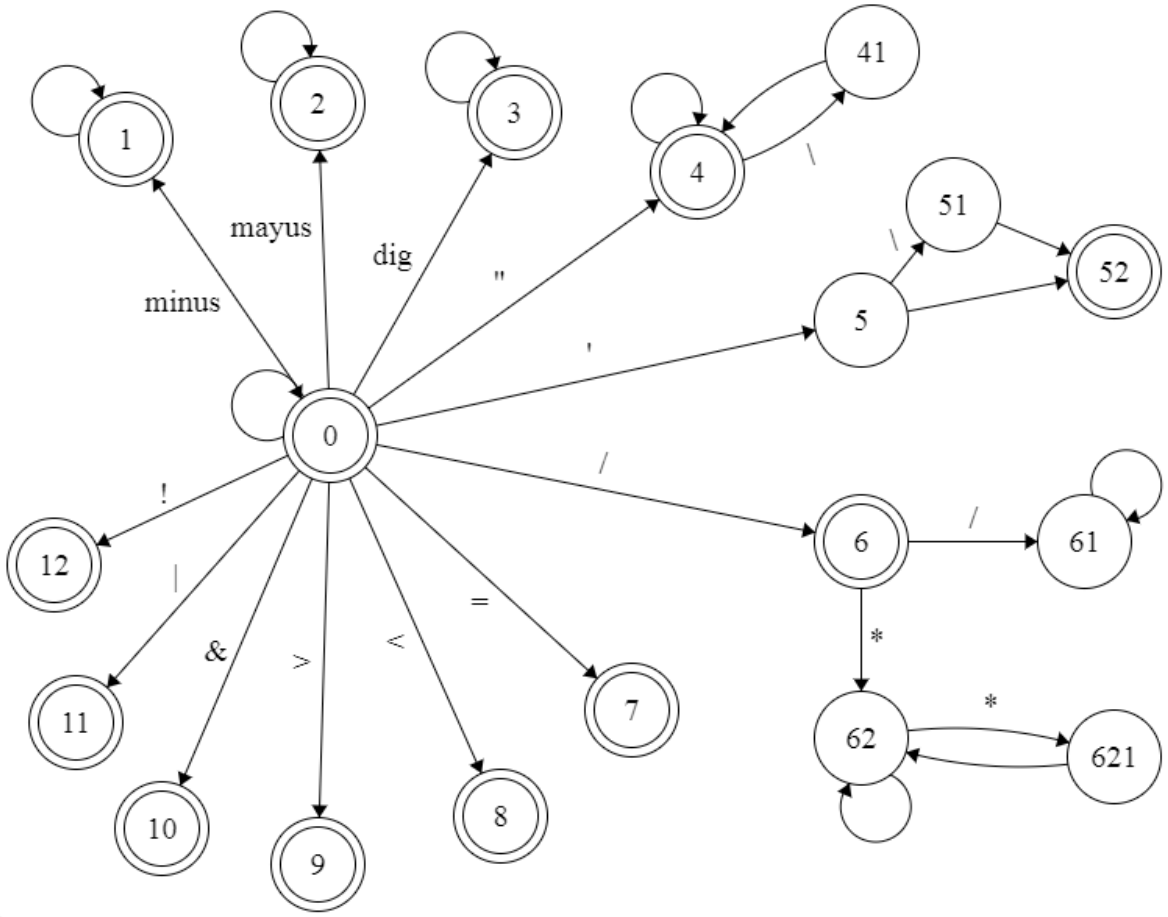
Tokens de palabras clave

El analizador es capaz de reconocer todas las palabras clave del lenguaje minijava. Cada palabra clave tiene su propio tipo de token distinto de los demás y su expresión regular es la misma palabra clave respetando mayúsculas y minúsculas.

Palabras claves:

String
class
extends
static
dynamic
boolean
char
int
public
private
void
null
if
else
while
return
this
new
true
false

Autómata



Estados:

0: Reconocedor de la gran mayoría de los tokens de un solo carácter (Ej: + - * ; . , () { } [])
1: Reconocedor de tokens tipo Identificador Metodo-Variable y de todas los tokens de palabras clave excepto String
2: Reconocedor de tokens tipo Identificador Clase y del token palabra clave String
3: Reconocedor de tokens de tipo Literal Entero
4: Reconocedor de tokens de tipo Literal Cadena de caracteres
41: Ignora el carácter de escape en tokens de tipo Literal Cadena de caracteres
5: Pertenece a los estados reconocedores de tokens de tipo Literal Caracter
51: Ignora el carácter de escape en tokens de tipo Literal Caracter
52: Reconocedor de tokens de tipo Literal Caracter
6: Reconocedor de tokens de tipo Operador División
61: Ignora los comentarios simples
62: Pertenece a los estados que ignoran comentarios multilinea
621: Pertenece a los estados que ignoran comentarios multilinea
7: Reconocedor de tokens de tipo Asignación Igual y Operador Igual
8: Reconocedor de tokens de tipo Operador Menor y Menor Igual
9: Reconocedor de tokens de tipo Operador Mayor y Mayor Igual
10: Reconocedor del token de tipo Operador AND
11: Reconocedor del token de tipo Operador OR
12: Reconocedor de tokens de tipo Operador Desigual y Negación

Errores léxicos detectados.

Caracter inesperado o inválido:

Ocurre cuando el analizador encuentra un carácter que no pertenece al alfabeto o cuando encuentra un carácter que si pertenece a dicho alfabeto pero se encuentra en una ubicación no esperada (por ejemplo un carácter $ fuera de un literal cadena de caracteres).

Literal cadena de caracteres mal formada:

Ocurre cuando un token de tipo literal cadena de caracteres no ha sido cerrado correctamente (por ejemplo se encontró un salto de linea, o el fin del archivo antes de la comilla doble que cierra el literal).

Literal caracter vacío:

Ocurre cuando un token de tipo literal carácter es vació, es decir cuando no se encuentran dos comillas simples seguidas (‘’) o un mal uso del carácter de escape (barra invertida \) (por ejemplo si se encuentra lo siguiente: ‘\’).

Literal caracter mal formado:

Ocurre cuando un token de tipo literal carácter no ha sido cerrado correctamente o cuando contiene mas de un único carácter dentro de las comillas (por ejemplo: ‘aa’ , ‘b , ‘\qq’).

Comentario multilinea sin cierre:

Ocurre cuando un comentario multilinea no es cerrado correctamente con los caracteres ‘*/’

Operador AND mal formado:

Ocurre cuando un token de tipo operador AND se encuentra mal formado, es decir luego del carácter & se encuentra otro carácter que no es & o es el fin del archivo.

Operador OR mal formado:

Ocurre cuando un token de tipo operado OR se encuentra mal formado, es decir, luego del carácter | se encuentra otro carácter que no es | o es el fin del archivo.

Testing

Para el testing del programa se utilizaron varios casos de prueba tanto correctos como incorrectos.
A continuación se listaran los nombres de los archivos de test junto con una breve descripción de su propósito y resultados esperados.

Casos de test correctos:

Test1.java

Este caso contiene todos los tokens validos del programa junto con comentarios simples y comentarios multilinea.
Se espera que el analizador reconozca todos tokens y que ignore todos los comentarios.

Test2.java

Este caso contiene un solo comentario simple luego de la primer linea en blanco
Se espera que el analizador ignore el comentario.

Test3.java

En este caso el archivo esta vacío.
Se espera que el analizador reconozca el token de fin de archivo

Test4.java

Este caso contiene varios tokens de tipo literal cadena de caracteres con utilizando el carácter de escape \ (barra invertida)
Se espera que el analizador ignore el carácter de escape y solo reconozca el carácter siguiente, también se espera que agregue un salto de linea o un tab según corresponda.

Test5.java

Este caso contiene varios tokens de tipo identificador de clase y de método-variable utilizando varios guiones bajos
Se espera que el analizador reconozca todos los tokens.

Test6.java

Este caso contiene varios comentarios simples y multilinea anidados y consecutivos
Se espera que el analizador ignore todos los comentarios.

Test7.java

Este caso contiene varios tokens de signos de puntuación y de operadores.
Se espera que el analizador reconozca todos los tokens.

Test8.java

Este caso contiene tokens de tipo literal cadena de caracteres con varios caracteres “extraños” entre las comillas separadoras que pueden ser parte del alfabeto o no.
Se espera que el analizador reconozca normalmente los caracteres parte del alfabeto que están dentro del token.
Para los caracteres extraños que no son parte del alfabeto se espera que sean convertidos según el criterio de la clase “BufferedReader” provista por Java que fue utilizada para leer el archivo de entrada y sean reconocidos como parte del token literal cadena de caracteres.

Test9.java

Este caso contiene varios tokens de tipo literal carácter con caracteres “extraños” dentro de las comillas simples y con algunos utilizando el carácter de escape
Se espera que el analizador reconozca todos los tokens.

Test10.java y Test11.java

Estos casos contienen el código fuente de la clase “EntradaSalida” y otras clases respectivamente
Se espera que el analizador reconozca todos los tokens e ignore los comentarios.

Casos de test incorrectos:

Test1.java

Este test contiene comentarios multilinea sin cerrar
Se espera que ocurra el error de comentario multilinea sin cerrar.

Test2.java

Este test contiene un carácter inesperado (forma parte del alfabeto)
Se espera que ocurra el error de carácter inesperado.

Test3.java

Este test contiene un token de tipo operador AND mal formado.
Se espera que ocurra el error de operador AND mal formado.

Test4.java y Test5.java

Estos test contienen tokens de tipo operador OR mal formados.
Se espera que ocurran los errores de operador OR mal formado.

Test6.java, Test7.java, Test8.java, Test9.java, Test10.java y Test11.java

Estos test contienen tokens de tipo literal carácter mal formados o vacíos.
Se espera que ocurran los errores de literal carácter vacíos o mal formados donde corresponda.

Test12.java y Test13.java

Estos test contienen tokens de tipo literal cadena de caracteres sin cerrar.
Se espera que ocurran los errores de literal cadena de caracteres sin cerrar.

Test14.java y Test15.java

Estos test contienen caracteres inesperados entre tokens de tipo Identificador Método-Variable.
Se espera que ocurran los errores de caracteres inesperados.