

# Informe Etapa 1

## Compiladores e Interpretes

Nahuel Ignacio Fuentes

August 29, 2023

### Uso

El proyecto fue desarrollado en IntelliJ y compilado con Java 11, para poder compilarlo es necesario usar la version de Java mencionada y compilar los archivos fuentes en un ejecutable.

Una vez con el ejecutable se puede utilizar el programa desde la terminal de comandos, en el lugar donde se encuentra el ejecutable del programa y los archivos a compilar se debe escribir :

```
java -jar Compilador.jar programa.java
```

Siendo *Compilador.jar* el ejecutable y *programa.java* el archivo a compilar.

Es muy importante recordar que ambos deben estar al mismo nivel para poder acceder de manera correcta, sino tambien se puede suministrar un camino hacia el archivo a compilar, siempre asumiendo que se parte de la ubicacion actual en la que se encuentra la terminal, se le pueden suministrar diversos archivos, mientras que estos existan y se encuentren en los caminos especificados como argumentos.

### Tokens y sus Expresiones regulares asociadas

Por simplificacion asociaremos a estas expresiones regulares las siguientes abreviaciones, con el fin de hacer la expresion resultante más legible :

$Digito = [0..9]$

$Letra\_Mayus = [A..Z]$

$Letra\_Minus = [a..z]$

$Letra = Letra\_Minus | Letra\_Mayus$

$Character = [Letra | Digito | , | . | ; | ( | . | ) ]$  , cualquier caracter valido

$Exponente = [ e | E ] [ + | - ] Digito^+$

- Identificador de clase  
 $idClase = Letra\_Mayus [Letra \mid Digito \mid \_ ]^*$
- Identificador de metodo y variable  
 $idMetVar = Letra\_Minus [Letra \mid Digito \mid \_ ]^*$

- Enteros  
 $intLiteral = Digito^n, 1 \leq n \leq 9$

- Floats  
 $floatLiteral = [ Digito^+ \cdot Digito^* [Exponente \mid ] ] \mid [ \cdot Digito^+ [Exponente \mid ] ] \mid [Digito^+ Exponente]$

Existe una restriccion de un valor maximo para este tipo de literales segun la definicion de Java, esta es que debe estar entre los valores de  $1.40e^{-45}$  y  $3.4028235e^{38}$  esto escapa a la expresi3n regular que tiene asociada y se verifica en la implementaci3n.

- Caracteres  
 $charLiteral = ' [[Character - \{ \backslash, ' \} ] \mid [ \backslash Character ] ] '$
- String  
 $stringLiteral = "[Character - \{ \backslash, ', \backslash n \} \mid [ \backslash " ] ] *"$
- Puntuaci3n  
 $openPar = ($   
 $closePar = )$   
 $openCurl = \{$   
 $closeCurl = \}$   
 $period = .$   
 $comma = ,$   
 $semiColon = ;$
- Operadores  
 $opAdd = +$   
 $opSub = -$   
 $opProd = *$   
 $opIntDiv = \%$   
 $opDiv = /$   
 $opLess = <$   
 $opLessEq = \leq$   
 $opEq = ==$   
 $opGreater = >$   
 $opGreaterEq = \geq$   
 $opAnd = \&\&$   
 $opOr = \parallel$   
 $opNot = !$

*opNotEq* = !=

- Asignacion

*assign* = =

*assignAdd* = + =

*assignSub* = - =

- Palabras reservadas

Tienen como id asociado *resWord*, estas no tienen asociadas ninguna Expresion Regular, lo mismo con las especificas para los tokens booleanos y nulo, *litBoolean* y *litNull*.

## Tipos de errores

Se reconocen diversos tipos de errores lexicos, estos son :

- Simbolos que no pertenecen al alfabeto
- Comentarios multilinea sin cerrar
- Cadena de caracteres sin cerrar
- Operadores logicos no validos
- Floats que exceder el valor representable
- Floats que estan mal formateados

## Clases utilizadas

Durante el desarrollo se pensó 4 Clases para la totalidad de la etapa, esas clases fueron LexicalAnalyzer, FileManager, Token y Main. Luego tambien se implementó una excepción específica llamada LexicalException para cuando se detectan excepciones lexicas, nos permite obtener información acerca de donde y porque ocurrió el error.

Además la clase FileManager paso a ser interfaz para poder probar distintas implementaciones, finalmente quedando la clase ImpFileManager que implementa la interfaz previamente mencionada a traves del uso de FileReader y BufferedReader.

La clase LexicalAnalyzer es la encargada de intentar reconocer tokens de un archivo y luego retornarlos o informar alguna excepción que haya ocurrido, estas por último las maneja la clase Main que se encarga de mostrar una gráfica para poder informar los tokens o los errores que ocurrieron, además de inyectar el manejador de archivos con el camino especificado por parametro al analizador.

## Logros intentados

Los logros que se intentaron resolver durante el desarrollo de la etapa fueron :

- Entrega Anticipada
- Imbatibilidad Lexica
- Reporte de Error Elegante
- Columnas
- Floats

## Consideraciones de diseño

Se opto por tener un único tipo de excepción lexica en vez de generar una herencia de excepciones a partir de ella, en su lugar se agrega una breve descripción del causante de su lanzamiento pero puede haber casos donde no representa la realidad, haber usado herencia habria hecho que aumente la cantidad de excepciones pero siendo una mejor opción por si se deseaba agregar más excepciones. Sin embargo seguian habiendo ambigüedad en que causo el error, si por ejemplo fue un caracter invalido o que se esperaba las comillas para cerrar un caracter.

Tambien se decidio, teniendo en cuenta que un literal entero es de 9 digitos, si se encuentra un numero mayor a 9 digitos no se toma como un error, sino que se corta al 9no digito y se retorna como un token valido y el resto de numero es otro entero separado.

Otro aspecto importante a mencionar es que el CR, Carriage Return, que lee el file manager se ve contemplado en el analizador lexico, esto puede que no funcione en otros,

Por ultimo se tomo la decision de marcar donde se esperaba el fin de un comentario multilinea si se encuentra que no se cerró, luego se lanza una excepción cuyo lexema asociado simplemente es el inicio del comentario multilinea.