Se desea construir un analizador sintáctico en JAVA, basado en la técnica LL(1), para reconocer y operar expresiones en notación prefija, es decir, expresiones en el que el operador precede a los dos operandos. En estas expresiones no es necesario utilizar paréntesis. A continuación, se exponen algunas de estas expresiones y el resultado de la operación:

Operación	Resultado
7	7
+ 3 4	7 = (3+4)
+ + 3 4 5	12 = (3+4)+5
+ * 3 8 + - 6 5 1	26 = (3*8)+((6-5)+1)
* - * 5 + 1 1 8 6	12 = ((5*(1+1))-8)*6
- 10 * - 5 3 4	2 = 10-(5-3)*4

Son incorrectas las expresiones que no pueden evaluarse como, por ejemplo:

Operación
7 8
3 4 +
+ 3 4 5 +
8 8 8 - -
* - * 5 + 1 1 8
+ - 10 * - 5 3 4

Se pide:

Apartado A (5 puntos)

Diseñar un programa en JAVA que tenga como entrada un fichero con una expresión en notación prefija y de cómo resultado CORRECTO en caso de que la expresión este bien formada (pertenezca al lenguaje) y ERROR en caso contrario.

Apartado B (1 puntos, sólo si el apartado A es correcto)

Extender el lenguaje para que reconozca entradas de la forma `nOper( <exp> )`, en cuyo caso el programa deberá devolver `NOPER=`<*número de operadores*>, es decir debe contar el número de operadores que hay en la entrada; o ERROR en caso de que la expresión no este correctamente formada. La siguiente tabla muestra algunos ejemplos:

Entrada	Salida
nOper (* - * 5 + 1 1 8 6)	NOPER=4
nOper (7)	NOPER=0
nOper (+ + 3 4 5)	NOPER=2
nOper (8 8 8 - -)	ERROR

Apartado C (2 puntos, sólo si el apartado A es correcto)

Extender el lenguaje para que reconozca entradas de la forma `maxProf( <exp> )`, en cuyo caso el programa deberá devolver `MAXPROF=<profundidad máxima>`, es decir debe analizar la expresión y obtener el valor máximo de la pila que hay que utilizar para efectuar la operación; o ERROR en caso de que la expresión no este correctamente formada. La siguiente tabla muestra algunos ejemplos:

Entrada	Salida
`maxProf( 7 )`	`MAXPROF=0`
`maxProf ( + + 3 4 5 )`	`MAXPROF=2`
`maxProf ( * - * 5 + 1 1 8 6 )`	`MAXPROF=4`
`maxProf ( + 8 * + + - 8 5 4 )`	`ERROR`

Apartado D (2 punto, sólo si el apartado A es correcto)

Extender el lenguaje para que reconozca entradas de la forma `eval( <exp> )`, en cuyo caso el programa deberá devolver `EVAL=<valor de la expresión>`, es decir debe realizar la operación y devolver el valor; o ERROR en caso de que la expresión no este correctamente formada. La siguiente tabla muestra algunos ejemplos:

Entrada	Salida
`eval( 7 )`	`EVAL=7`
`eval( + 3 4 )`	`EVAL=7`
`eval(* - * 5 + 1 1 8 6)`	`EVAL=12`
`eval( 1 2 3 + * )`	`ERROR`

NOTAS:

Para completar las especificaciones y responder a cualquier duda sobre el funcionamiento del programa que se pide construir, se proporciona un programa de ejemplo ya compilado que realiza exactamente las funciones que se piden, y un conjunto de casos de prueba. Este programa se encuentra en un fichero comprimido para distintos sistemas operativos. En Linux, una vez descomprimido será necesario modificar los permisos de ejecución mediante la instrucción:

```
chmod +x parser
```

La ejecución del programa, se realiza mediante la instrucción:

`./parser` *<fichero de entrada>*

Para facilitar la construcción del analizador se proporciona el analizador léxico denominado `Lex.lex` para JFLEX, que devuelve los tokens representados como números enteros. Para obtener el analizador léxico debe ejecutarse la instrucción:

```
jflex Lex.lex
```

También se proporciona un esquema de la clase que implementará el analizador sintáctico denominado `Parser.java` que debe completarse con la implementación

necesaria de manera que pueda compilarse y ejecutar con la siguiente secuencia de instrucciones:

```
javac Parser.java
java Parser <fichero de entrada>
```

obteniendo la misma salida que se obtendría con el programa de ejemplo.

Solo debe enviarse el fichero `Parser.java` por lo que cualquier modificación de cualquier otro fichero no tendrá efecto.