|  |
| --- |
| **Diseño del analizador léxico, del gestor de errores y de la tabla de símbolos.** |
| [Escribir el subtítulo del documento] |
|  |
|  |
|  |
| Grupo 1 |
| **30/11/2011** |
|  |

Contenido

[Componentes del grupo 3](#_Toc311668984)

[Identificación de tokens 4](#_Toc311668985)

[Gramática del analizador léxico 6](#_Toc311668986)

[Notación “habitual” 6](#_Toc311668987)

[Notación EBNF 7](#_Toc311668988)

[Autómata finito 10](#_Toc311668989)

[Números 10](#_Toc311668990)

[Operadores 11](#_Toc311668991)

[Resto 13](#_Toc311668992)

[Acciones semánticas 14](#_Toc311668993)

[Reconocimiento de números 14](#_Toc311668994)

[Reconocimiento de operadores 14](#_Toc311668995)

[Tabla de transiciones del autómata 18](#_Toc311668996)

[Tabla de símbolos 22](#_Toc311668997)

[Gestión de palabras reservadas. 22](#_Toc311668998)

[Gestión de ámbitos. 22](#_Toc311668999)

[Gestión de los errores 23](#_Toc311669000)

[Funcionamiento básico 23](#_Toc311669001)

[Tipos 23](#_Toc311669002)

[Interfaz 23](#_Toc311669003)

[Tabla de errores léxicos 24](#_Toc311669004)

[Implementación analizador léxico 25](#_Toc311669005)

[Diagrama de clases 25](#_Toc311669006)

[Prototipo de la tabla de símbolos 26](#_Toc311669007)

[Prototipo del gestor de errores 27](#_Toc311669008)

# Componentes del grupo

* Alina Gheorghita
* Cristina García
* Pilar Torralbo
* Tomás Restrepo
* Guillermo José Hernández
* Laura Reyero

# Identificación de tokens

Clasificamos los tokens que reconocerá nuestro analizador según el tipo al que pertenezcan, de la siguiente manera:

|  |  |
| --- | --- |
| **Tipo de token** | **Codificación** |
| EOF | (FIN, null) |
| LIT\_CARACTER | (LIT\_CARACTER, [caracteres]) |
| LIT\_CADENA | (LIT\_CADENA, [índice tabla]) |
| CADENA | (CADENA, [puntero a tabla]) |
| NUM\_REAL | (NUM\_REAL, [valor real]) |
| NUM\_ENTERO | (NUM\_ENTERO, [valor entero]) |
| SEPARADOR | (SEPARADOR, [enumerado asociado]) |
| OP\_ARITMETICO | (OP\_ARITMETICO, [enumerado asociado]) |
| OP\_COMPARACION | (OP\_COMPARACION, [enumerado asociado]) |
| OP\_LOGICO | (OP\_LOGICO, [enumerado asociado]) |
| OP\_ASIGNACION | (OP\_ASIGNACION, [enumerado asociado]) |
| COMENT\_LINEA | (COMENTARIO,[puntero a tabla]) |
| COMENT\_LARGO | (COMENTARIO,[puntero a tabla]) |
| ERROR | (ERROR, null) |

**LIT\_CARACTER:** secuencia de caracteres que empieza y acaba por comillas simples

**LIT\_CADENA:** secuencia de caracteres que empieza y acaba por comillas dobles

**CADENA:** secuencia de caracteres que determinará los identificadores y palabras reservadas del analizador.

**NUM\_REAL**: número real.

**NUM\_ENTERO**: número entero.

**SEPARADOR**: enumerado “.” | ";" | "{" | "}" | "[" | "]" | "#" | "##" | "(" | ")" | "<:" | ":>" | "<%" | "%>" | "%:" | "%:%:"

**OP\_ARITMETICO**: enumerado "+" | "-" | "++" | "--" | "\*" | "/" |  "%"

**OP\_COMPARACION:** enumerado"==" | "!=" | "<" | ">" | "<=" | ">="

**OP\_LOGICO**: enumerado "&&" | "||" | "!" | "&" | "|" | "~" | "ˆ " | "<<" | ">>" | "and" | "and\_eq" | "bitand" | "bitor" | "compl" | "not" | "not\_eq" | "or" | "or\_eq" | "xor" | "xor\_eq"

**OP\_ASIGNACIÓN:** enumerado "=" | "+=" | "-=" | "\*=" | "/=" | "%=" |  "ˆ=" | "&=" | "|=" | ">>=" | "<<=" | “->”

**COMENT\_LINEA**: Comentario que termina con el fin de línea.

**COMENT\_LARGO:** Comentario largo, que puede ser de una o varias líneas de longitud, y que al igual que el comentario de línea se tiene en cuenta solo por tratarse de un traductor y no de un compilador habitual.  
  
**ERROR:** Error general de tipo léxico, con atributo nulo. El error particular se pasa únicamente al gestor de errores, quien al final del todo lo imprimirá por pantalla.

# Gramática del analizador léxico

Notación “habitual”  
  
literal → numero | cadCar | litBooleano | litString | litPuntero

litPuntero → nullptr

litString → “((CajonDesastre - {“,\,\n}) | secEscape | nombredecaracteruniversal})+ ”

litBooleano →  true | false

cadCar → ‘ (CajonDesastre - {‘, \, \n, secEscape, nombredecaracteruniversal})+ ‘

nombredecaracteruniversal → \u cuartetoHex | \U cuartetoHex cuartetoHex

cuartetoHex → digHex digHex digHex digHex

secEscape →  secSimpleEsc | secOctalEsc | secHexEsc

secSimpleEsc → \ , ‘ | \* | ? | \ | a | b | f | n | r | t | v

secOctalEsc → \ , octal | octal octal | octal octal octal

secHexEsc →  \x digHex+

numero → entero | real

entero → (decimal | hexadecimal | octal), [sufEntero]

hexadecimal → 0x | 0X , digitoHex+

octal → 0, digOct

decimal →  digito | digSinCero digito\* [exponente]

sufEntero → (sufSinSigno [sufijoLargo]) | (sufSinSigno [sufijoLargoLargo]) |

(sufLargo [sufijoSinSigno]) | (sufLargoLargo [sufijoSinSigno])

sufijoSinSigno → u | U

sufijoLargo →      l | L

sufijoLargoLargo → ll | LL

real → **.** digito**+**  [exponente] [sufReal] | digito**+ .** digito\*  [exponente] [sufReal] |

digito**+** exponente [sufReal]

exponente → e | E , [+ | -], digito**+**

sufReal → f | l | F | L

identificador → noDigito (noDigito | digito)\*

digOct → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

digHex → digito | A | B | C | D | E | F | a | b | c | d | e | f

noDigito → “\_” | letra

cajonDesastre → letra | digito | “\_” | “{“ | “}” | “[“ | “]” | “#” | “(“ | “)” | “<“ | “>“ | “%” | “:” | “;” | “.” | “?” | “\*” | “+” | “-” | “/” | “^” | “&” | “|” | ““ | “!” | “=“ | “,” | “\” | “"“ | “’”

letra →  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" |"R" | "S" | "T" | "U"  | "V" | "W" | "X" | "Y" | "Z" |  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u"  | "v" | "w" | "x" | "y" | "z"

digito → digSinCero | 0

digSinCero → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

DELIM → ‘ ’ (blanco) | TAB | EOL | EOF | ‘;’ | ‘|’ | ‘:’ | ‘+’ | ‘-’ | ‘/’ | ‘\*’ | ‘<’ | ‘>’ | ‘=’ | ‘&’ | ‘^’ | ‘%’ | ‘!’ | ‘~’ | ‘{‘ | ‘}’ | ‘[‘ | ‘]’ | ‘#’ | ‘(‘ | ‘)’

DELIM2 → DELIM | digito | noDigito

Notación EBNF  
  
literal = numero | cadCar | litBooleano | litString | litPuntero ;

litPuntero = nullptr ;

litString = ‘ “ ‘ , ( (cajonDesastre2 | secEscape | nombredecaracteruniversal),

{ (cajonDesastre2 | secEscape | nombredecaracteruniversal}, ‘ ” ‘ ;

litBooleano = true | false ;

cadCar = “ ‘ “,  (CajonDesastre - {“ ‘ “, “\”, “\n”, secSalida, nombredecaracteruniversal}),

{CajonDesastre - {‘,\,\n,secEscape,nombredecaracteruniversal}}, “ ‘ “ ;

nombredecaracteruniversal = ‘\u’, cuartetoHex | ‘\U’, cuartetoHex, cuartetoHex ;

cuartetoHex = digHex, digHex, digHex, digHex ;

secEscape = simpleSeq | octalSeq | hexSeq ;

secSimpleEsc = \ , ‘ | \* | ? | \ | a | b | f | n | r | t | v ;

secOctalEsc = \ , octal | (octal, octal) | (octal, octal, octal) ;

secHexEsc =  ‘\x’, digHex, {digHex} ;

numero = entero | real ;

entero = (decimal | hexadecimal | octal),  [sufEntero] ;

hexadecimal = ‘0x’ | ‘0X’ , digHex, {digHex} ;

octal = 0, 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 ;

decimal =  digito | digSinCero, {digito}, [exponente] ;

sufEntero = (sufSinSigno [sufijoLargo]) | (sufSinSigno [sufijoLargoLargo]) |

(sufLargo [sufijoSinSigno]) | (sufLargoLargo [sufijoSinSigno]) ;

sufSinSigno = “u” | “U” ;

sufLargo = “l” | “L” ;

sufLargoLargo = ‘ll’ | ‘LL’ ;

real = ((“.” , digito, {digito}) | (digito, {digito} , “.” ,  {digito}),  [exponente])

| (digito, {digito}, exponente), [sufReal] ;

exponente = e | E , [+ | -], digito, {digito} ;

sufReal = “f” | “l” | “F” | “L” ;

identificador = noDigito, (noDigito | digito)\* ;

digHex = digito | A | B | C | D | E | F | a | b | c | d | e | f ;

noDigito = “\_” | letra ;

cajonDesastre = letra | digito | “\_” | “{“ | “}” | “[“ | “]” | “#” | “(“ | “)” | “<“ | “>“ | “%” | “:” | “;” | “.” | “?” | “\*” | “+” | “-” | “/” | “^” | “&” | “|” | ““ | “!” | “=“ | “,” | “\” | “"“ | “’” ;

cajonDesastre2 = letra | digito | “\_” | “{“ | “}” | “[“ | “]” | “#” | “(“ | “)” | “<“ | “>“ | “%” | “:” | “;” | “.” | “?” | “\*” | “+” | “-” | “/” | “^” | “&” | “|” | ““ | “!” | “=“ | “,” | “’” ;

letra =  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U"  | "V" | "W" | "X" | "Y" | "Z" |  "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u"  | "v" | "w" | "x" | "y" | "z" ;

digito = digSinCero | 0 ;

digSinCero = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;

DELIM =‘ ’ (blanco) | TAB | EOL | EOF | ‘;’ | ‘|’ | ‘:’ | ‘+’ | ‘-’ | ‘/’ | ‘\*’ | ‘<’ | ‘>’ | ‘=’ | ‘&’ | ‘^’ | ‘%’ | ‘!’ | ‘~’ | ‘{‘ | ‘}’ | ‘[‘ | ‘]’ | ‘#’ | ‘(‘ | ‘)’

DELIM2 = DELIM | digito | noDigito ;

# Autómata finito

## Números

## Operadores





## Resto



# Acciones semánticas

Reconocimiento de números  
  
**INIC**: Código de inicialización de variables  
  
**A**: Lee el siguiente carácter de la entrada (código fuente)  
  
**\*** : Retrocede una posición el puntero de lectura del fichero fuente (devuelve el último carácter leído a la entrada)  
  
**B**: Genera un token de número entero para el analizador sintáctico  
  
**C**: Conversión y adición del carácter de preanálisis al valor acumulado de la parte decimal del número.  
  
**D1**: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número. (base 10)  
  
**D2**: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número. (base 8)  
  
**D3**: Conversión y adición del carácter de preanálisis al valor acumulado de la parte entera del número. (base 16)  
  
**E:** Conversión y adición del carácter de preanálisis al valor acumulado de la parte exponencial del número  
  
**S**: Guarda el signo de la E para los enteros (por ejemplo un 1 o un -1).  
  
**F**: Genera un token de número real para el analizador sintáctico.

{

token = GeneraToken(NUM\_REAL, parteEntera + parteDecimal + parteExponencial \* signo + sufijoReal);

return token;

}

**G**: Genera un token de número entero para el analizador sintáctico

{

token = GeneraToken(NUM\_ENTERO,parteEnteraDecimal + parteEnteraOctal + parteEnteraHexadecimal + sufijoEntero);

return token;

}

Reconocimiento de operadores  
  
**H1**: Genera un token de operador de **asignación “=”** para el analizador sintáctico.

{

token = GeneraToken (OP\_ASIG, ASIG); //Valor del enumerado ASIG, si decidimos juntar los 3 tipos de operadores de asignacion en uno

return token;

}

**H2**: Genera un token de operador de **asignación con suma “+=”** para el analizador sintáctico.

{

token = GeneraToken (OP\_ASIG, ASIGSUM);

return token;

}

**H3**: Genera un token de operador de **asignación con resta “-=”**  para el analizador sintáctico.  
**H4**: Genera un token de operador de **asignación con multiplicacion ”\*=”**  para el analizador sintáctico.  
**H5**: Genera un token de operador de **asignación con division ”/=”**  para el analizador sintáctico.  
**H6**: Genera un token de operador de **asignación con modulo “%=”**  para el analizador sintáctico.  
**H7**: Genera un token de operador de **asignación con and “&=”** para el analizador sintáctico.  
**H8**: Genera un token de operador de **asignación con or  “|=”** para el analizador sintáctico.  
**H9**: Genera un token de operador de **asignación con xor “ˆ=”**  para el analizador sintáctico.  
**H10**: Genera un token de operador de **asignación “>>=”**para el analizador sintáctico.  
  
**H11**: Genera un token de operador de **asignacion “<<=”**  para el analizador sintáctico.  
  
-------------------------------------------------------------------------------------------------------------------  
**I1**: Genera un token de operador **aritmetico suma “+”** para el analizador sintáctico.

{

token = GeneraToken (OP\_ARIT, SUMA); //Valor del enumerado SUMA

return token;

}

**I2**: Genera un token de operador **aritmetico resta “-”** para el analizador sintáctico.  
**I3**: Genera un token de operador **aritmetico multiplicacion “\*”** para el analizador sintáctico.  
**I4**: Genera un token de operador **aritmetico division “/”** para el analizador sintáctico.  
**I5**: Genera un token de operador **aritmetico modulo “%”** para el analizador sintáctico.  
**I6**: Genera un token de operador **aritmetico “++”** para el analizador sintáctico.  
**I7**: Genera un token de operador **aritmetico “--”** para el analizador sintáctico.  
  
----------------------------------------------------------------------------------------------------------------------  
**J1**: Genera un token de operador **comparación igual “==”** para el analizador sintáctico.

{

token = GeneraToken (OP\_COMP, IGUAL); //Valor del enumerado SUMA

return token;

}

**J2**: Genera un token de operador **comparación distinto “!=”** para el analizador sintáctico.  
**J3**: Genera un token de operador **comparación menor “<”** para el analizador sintáctico.  
**J4**: Genera un token de operador **comparación mayor “>”** para el analizador sintáctico.  
**J5**: Genera un token de operador **comparación menorIgual “<=”** para el analizador sintáctico.  
**J6**: Genera un token de operador **comparación mayorIgual “>=”** para el analizador sintáctico.  
  
------------------------------------------------------------------------------------------------------------------  
  
**K1**: Genera un token de operador **lógico  "&&"**  para el analizador sintáctico.

{

token = GeneraToken (OP\_LOG, AND); //Valor del enumerado SUMA

return token;

}

**K2**: Genera un token de operador **lógico "||"**  para el analizador sintáctico.  
**K3**: Genera un token de operador **lógico "!"**  para el analizador sintáctico.  
**K4**: Genera un token de operador **lógico "&"**  para el analizador sintáctico.  
**K5**: Genera un token de operador **lógico  "|"**  para el analizador sintáctico.  
**K6**: Genera un token de operador **lógico "~"** para el analizador sintáctico.  
**K7**: Genera un token de operador **lógico "ˆ "**  para el analizador sintáctico.  
**K8**: Genera un token de operador **lógico "<<"** para el analizador sintáctico.  
**K9**: Genera un token de operador **lógico ">>"** para el analizador sintáctico.  
  
-----------------------------------------------------------------------------------------------------------  
**M1**: Genera un token separador **punto y coma “;”** para el analizador sintáctico.

{

token = GeneraToken (SEP, PUNTCOMA);

return token;

}

**M2**: Genera un token separador **corchete abierto “[“** para el analizador sintáctico.  
**M3**: Genera un token separador **corchete cerrado “]”** para el analizador sintáctico.  
**M4**: Genera un token separador **punto y coma “;”** para el analizador sintáctico.  
**M5**: Genera un token separador **almohadilla “#”** para el analizador sintáctico.  
**M6**: Genera un token separador **almohadilla doble “##”** para el analizador sintáctico.  
**M7**: Genera un token separador **paréntesis abierto “(“** para el analizador sintáctico.  
**M8**: Genera un token separador **paréntesis cerrado “)”** para el analizador sintáctico.  
**M9**: Genera un token separador **llave abierta “{”** para el analizador sintáctico.  
**M10**: Genera un token separador **llave cerrada “}”** para el analizador sintáctico.  
  
Reconocimiento del resto  
  
**L**: Añade el carácter leído al final del lexema ya acumulado:

{ Concatena(lexema, preanalisis); }

**M**: Búsqueda del lexema en la tabla de palabras reservadas  y si lo encuentra devuelve el token, si no lo encuentra búsqueda/inserción en la tabla de símbolos y generación del token para el analizador sintáctico:  
{  
        token = TS.getPalRes.Busca(lexema);  
        if(token != null)  
                    return token;

token = TS.Busca(lexema);

if (token == NULL)  {

token = GeneraToken(ID, TS.Inserta(lexema));

return token;

        }  
}  
  
**N**: Añade el carácter leído al final de la cadena con comillas dobles ya acumulado:

{ Concatena(cadenaComDobles(litString), preanalisis); }

**O**: Añade el carácter leído al final de la lexema cadena con comillas simples ya    acumulado:  
{ Concatena(cadenaComSimples(cadCar), preanalisis); }  
  
**P** : Genera un token de fin de la entrada para el analizador sintáctico:  
{

token = GeneraToken (FIN, NULL);

return token;

}  
  
**Q**: Se queda con la parte calculada en base decimal  
  
**R1**: Genera un token de tipo LIT\_CADENA  
  
**R2**: Genera un token de tipo LIT\_CARACTER  
  
**R3**: Genera un token flecha (para punteros)

**R4**: Genera un token punto  
  
**T**: Incrementa el número de líneas

# Tabla de transiciones del autómata

# Tabla de símbolos

Para C++ los ámbitos se definen al abrir llave ({) y cerrar llave (}) y en cualquier zona que se comporte como ámbito, se podrán declarar variables (bucles for, while, registros, case… ).

Como decidimos implementar la práctica en Java, podemos usar sus objetos predefinidos, como  hash tables, hash map, enum map, listas dinámicas, etc para la gestión de la tabla de símbolos.

## Gestión de palabras reservadas.

Dado que las palabras clave son también palabras reservadas, se van a almacenar dentro de la tabla de símbolos al arrancar el compilador.

Vamos a tener una tabla de palabras reservadas donde el campo clave es la cadena de caracteres que identifica a la palabra reservada. Como campo valor almacenaremos un valor numérico que identifique a cada una de las palabras.

## Gestión de ámbitos.

Vamos a gestionar nuestra tabla de símbolos con una tabla separada para cada ámbito. Tendremos una tabla de símbolos global, en la que almacenaremos cada identificador encontrado en ese ámbito junto con su tipo, así como una entrada para cada nuevo ámbito que aparezca en el código. Esta entrada tendrá información sobre el número de argumentos y su tipo, la forma de pasar los argumentos, el tipo de retorno y un puntero a la propia tabla de ese ámbito donde estará almacenada la información que él contiene.

En cada una de las tablas asociadas a los ámbitos, aparecerá como último atributo, un puntero que señalará cual es su ámbito padre.

Tendremos un puntero BLOQUE\_ACTUAL para facilitar la búsqueda dentro de la tabla de símbolos, y que como su propio nombre indica, apuntará al bloque actual.

# Gestión de los errores

## Funcionamiento básico

* Sustituirá a la gestión de errores básica en cualquier generador léxico-sintáctico como Javacc.
* Debe contabilizar el número de errores Léxicos y el número de errores sintácticos.
* Ofrecerá métodos distintos para insertar errores léxicos y sintácticos.
* Un método para imprimir los errores finales.
* Dos tablas distintas para léxico y sintáctico, cada una con la tabla de errores creada por las distintas transiciones incorrectas, mas errores gramaticales distintos en el sintáctico.
* Informativo, no soluciona errores.

## Tipos

Dos tipos distintos:

* Error Léxico
* Error Sintáctico

Principalmente se distinguen en que acceden a dos tablas distintas de errores y que el léxico guarda el carácter que produce el error, mientras que el sintáctico da linea y tipo de error (mucho mejor distinguidos).

## Interfaz

* **void** i**nsertarErrorLex:** Requiere el carácter que ha dado el error, número de linea, número de error. Inserta en una lista un error de tipo Léxico.

* **void** i**nsertaErrorSin:** Inserta en una lista un error de tipo sintáctico.
* **int** i**mprimeErrores:**
  + Devuelve un integer: 0 si no han ocurrido errores hasta el momento, 1 en otro caso.
  + Imprime los errores ocurridos de la forma:

“No hubo errores léxicos”

“5 errores léxicos:

“Linea 5. Llego un \* se esperaba (lista de valores válidos según el error de la tabla)

Linea 6...”

La impresión de errores sintácticos incluso mensajes de error más variados: Identificador no declarado, errores de tipos etc.

## Tabla de errores léxicos

Asigna distintos String que describen errores léxicos a números de entrada



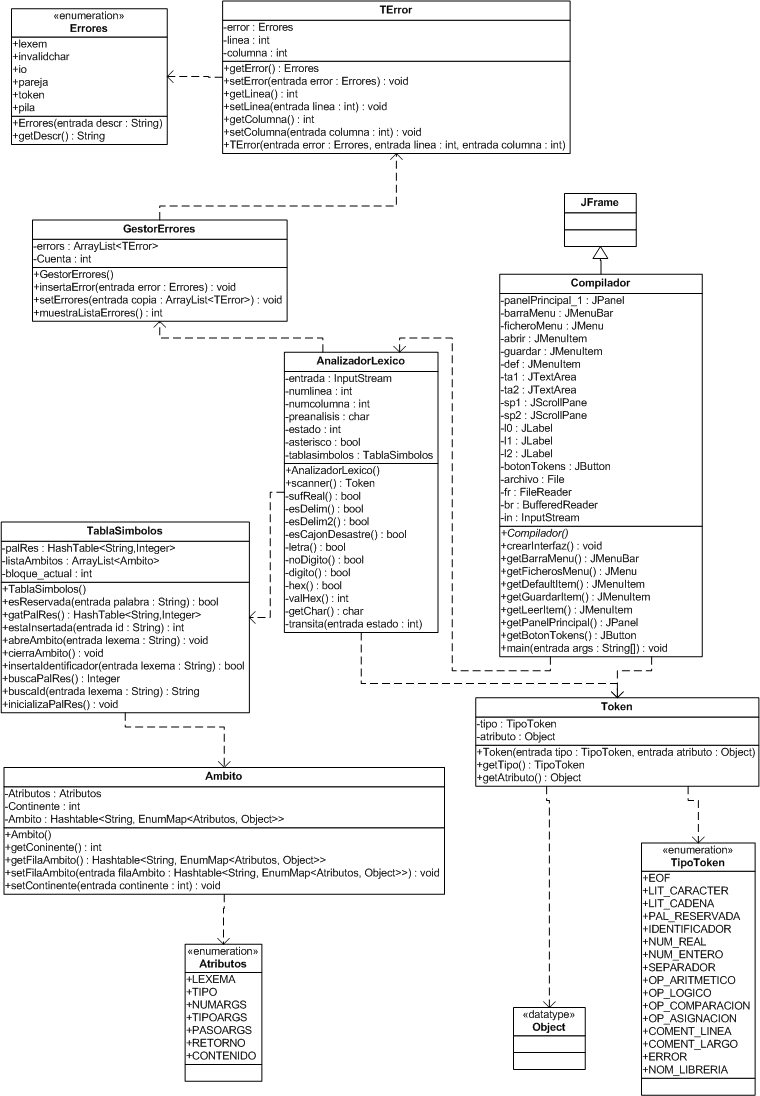
# Implementación analizador léxico

Hemos implementado el analizador léxico manualmente, sin servirnos de herramientas generadoras.

Cargamos el código a analizar como un InputStream, que vamos leyendo mediante caracteres y formando tokens uno a uno.

El estándar utilizado será UTF-8.

## Diagrama de clases



# Prototipo de la tabla de símbolos

En la tabla de símbolos almacenamos las palabras reservadas, en una hashtable, cuya clave es el string correspondiente a la palabra reservada y como valor, le asignamos un número entero.

En este caso trataremos algunas palabras de algunas librerías como palabras reservadas; es el caso de “cin”, “cout” y “string” por ejemplo.

La tabla de símbolos estará implementada como una lista doblemente enlazada en la que cada ámbito está conectado con su continente y su contenido. Esto será posible gracias a una estructura con dos punteros señalando al ámbito anterior (padre) y al ámbito siguiente (hijo).

Algoritmos

# Prototipo del gestor de errores

A medida que el analizador léxico va reconociendo los tokens del código de entrada, al encontrar un error léxico, creamos un token ERROR.

Desde el Gestor de errores vamos a ir introduciendo todos los tokens erróneos en una lista, añadiéndoles una descripción y el número de línea donde se encuentran.

Al finalizar el análisis léxico, se invocará al método que imprime por pantalla los errores producidos durante el análisis.