|  |
| --- |
| **Traductor de C++ a Pascal** |
| [Escribir el subtítulo del documento] |
|  |
|  |
|  |
| Grupo 1 |
| **16/12/2011** |
|  |

Contenido

[Componentes del grupo 4](#_Toc311756985)

[Comparativa de herramientas que generan analizadores 5](#_Toc311756986)

[Generadores de analizadores léxicos (scanner generators) 5](#_Toc311756987)

[Lex 5](#_Toc311756988)

[JLex 5](#_Toc311756989)

[Flex 5](#_Toc311756990)

[JFlex 6](#_Toc311756991)

[Ragel 7](#_Toc311756992)

[Comparativa de generadores léxicos 8](#_Toc311756993)

[Generadores de analizadores léxicos y sintácticos (scanner and parser generators) 9](#_Toc311756994)

[Antlr 9](#_Toc311756995)

[CookCC 9](#_Toc311756996)

[Grammatica 9](#_Toc311756997)

[JavaCC 10](#_Toc311756998)

[YaCC 10](#_Toc311756999)

[Comparativa de generadores léxicos y sintácticos 11](#_Toc311757000)

[Conclusiones 12](#_Toc311757001)

[Diseño del analizador léxico, del gestor de errores y de la tabla de símbolos. 14](#_Toc311757002)

[Identificación de tokens 14](#_Toc311757003)

[Gramática del analizador léxico 16](#_Toc311757004)

[Notación “habitual” 16](#_Toc311757005)

[Notación EBNF 18](#_Toc311757006)

[Autómata finito 20](#_Toc311757007)

[Números 20](#_Toc311757008)

[Operadores 21](#_Toc311757009)

[Resto 23](#_Toc311757010)

[Acciones semánticas 24](#_Toc311757011)

[Reconocimiento de números 24](#_Toc311757012)

[Reconocimiento de operadores 25](#_Toc311757013)

[Reconocimiento del resto 27](#_Toc311757014)

[Tabla de transiciones del autómata 28](#_Toc311757015)

[Tabla de símbolos 29](#_Toc311757016)

[Gestión de palabras reservadas. 29](#_Toc311757017)

[Gestión de ámbitos. 29](#_Toc311757018)

[Gestión de los errores 30](#_Toc311757019)

[Funcionamiento básico 30](#_Toc311757020)

[Tipos 30](#_Toc311757021)

[Interfaz 30](#_Toc311757022)

[Tabla de errores léxicos 31](#_Toc311757023)

[Implementación del analizador léxico y de los prototipos de la tabla de símbolos y del gestor de errores. 32](#_Toc311757024)

[Consideraciones previas a la implementación 32](#_Toc311757025)

[Implementación analizador léxico 33](#_Toc311757026)

[Diagrama de clases 34](#_Toc311757027)

[Prototipo de la tabla de símbolos 35](#_Toc311757028)

[Prototipo del gestor de errores 36](#_Toc311757029)

[Glosario 37](#_Toc311757030)

[Bibliografía 38](#_Toc311757031)

# Componentes del grupo

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

# Comparativa de herramientas que generan analizadores

## 

## Generadores de analizadores léxicos (scanner generators)

### Lex

Tipo de analizador generado: léxico.

Código generado: C

Plataforma: Unix

Se puede usar con: Yacc

Licencia: software propietario

Otras versiones: Flex (software libre)

Entrada: una tabla de expresiones regulares y fragmentos correspondientes del programa

Lex ayuda a escribir programas cuyo control de flujo es dirigido por instancias de expresiones regulares en el flujo de entrada. Es muy adecuado para las transformaciones de tipo script y para la segmentación de la entrada en la preparación de una rutina de análisis.

El reconocimiento de las expresiones es realizado por un autómata finito determinista generado por Lex. Los fragmentos de programa escrito por el usuario se ejecutan en el orden en el que las expresiones regulares correspondientes aparecen en el flujo de entrada.

### JLex

Tipo de analizador generado: léxico.

Código generado: Java

Plataforma: Multiplataforma.

Se puede usar con: CUP.

Licencia: Open Source y GPL-Compatible

Otras versiones: Jflex

### Flex

Tipo de analizador generado: léxico.

Código generado: C, C++ (flex++)

Plataforma: Unix

Se puede usar con: Yacc, Bison.

Licencia: software libre (BSD license).

Complejidad del analizador generado: O(n) sobre la longitud de entrada.

Un analizador léxico Flex normalmente tiene complejidad O(n) sobre la longitud de la entrada. Es decir, se realiza un número constante de operaciones para cada símbolo de entrada. Esta constante es bastante baja (GCC genera 12 Instrucciones para el bucle de reconociemiento del ATN). Nótese que la constante es independiente de la longitud del token, de la longitud de la expresión regular y del tamaño del ATN.

Sin embargo, una característica opcional de Flex puede causar la generación de un analizador con complejidad no lineal: el uso de la macro REJECT en un analizador con el potencial de reconocimiento de tokens muy largos. En este caso, el programador le pide explícitamente a flex "volver atrás e intentarlo de nuevo" después de que haya acertado alguna entrada. Esto causará que el DFA use backtracking para encontrar otros estados finales. En teoría, la complejidad en tiempo es O(n+m^2)\ge O(m^2) donde m es la longitud más larga del token (esto vuelve a ser O(n) si los tokens son "pequeños" con respecto al tamaño de la entrada). La función de REJECT no está habilitada por defecto, y sus implicaciones en el rendimiento están ampliamente documentadas en el manual de Flex .

### JFlex

Tipo de analizador generado: léxico.

Código generado: Java.

Plataforma:  Multiplataforma (debe ejecutarse en una plataforma que soporte JRE/JDK 1.1 ó superior).

Se puede usar con: CUP, BYacc/J, ANTLR.

Licencia: Software libre. (GPL)

Entrada: Un archivo con extensión “.flex” que incluya las expresiones regulares del lenguaje que se quiere reconocer.

JFlex es un generador de analizador léxico, para Java, escrito en Java. También es una reescritura de la herramienta muy útil JLex.

JFlex está diseñado para trabajar junto con el generador de analizador sintáctico LALR, CUP de Scott Hudson, y la modificación de Java de Berkeley Yacc BYacc/J de Bob Jamison. También se puede utilizar junto con otros generadores de analizadores sintácticos como ANTLR o como una herramienta independiente.

La lista de características se puede encontrar en su página web:

<http://jflex.de/features.html>

### Ragel

Tipo de analizador generado: léxico.

Código generado: Java, C, C++, Objective-C, D.

Plataforma: Multiplataforma.

Se puede usar con:

Licencia: Software libre. (Licencia GNU)

Entrada: Una tabla con expresiones regulares y fragmentos de código

Ragel es un compilador de máquina de estados finitos con soporte de salida para C, C #, Objective-C, D, Java, Go y el código fuente de Ruby. Es compatible con la generación de una tabla o control de flujo impulsado por las máquinas de estado de las expresiones regulares y / o gráficos de estado y también se puede construir analizadores léxicos a través del método “longest-match”.

Una característica única de Ragel es que las acciones del usuario pueden estar asociadas con las transiciones de una máquina de estados arbitraria usando los operadores que están integrados en las expresiones regulares. Ragel también es compatible con la visualización de la máquina generada a través de graphviz.

## Comparativa de generadores léxicos

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nombre** | **Autómata reconocido** | **Entrada** | **Lenguaje de salida** | **Licencia** |
| Flex | AFD(table driven) | Mixto | C | BSD |
| Lex | AFD(table driven) | Mixto | C | Propietario, CDDL |
| JLex | AFD | Mixto | Java | BSD-like |
| JFlex | AFD | Mixto | Java | GNU GPL |
| Ragel | AFD | Mixto | C,C++,D, Java, Objective-C, Ruby | GNU GPL |

## Generadores de analizadores léxicos y sintácticos (scanner and parser generators)

### Antlr

Analizadores generados: léxico y sintáctico.

Código generado: [C](http://en.wikipedia.org/wiki/C_(programming_language)), [C#](http://en.wikipedia.org/wiki/C_Sharp_(programming_language)), [Java](http://en.wikipedia.org/wiki/Java_(programming_language)), JavaScript, Ada95, Python, Perl

Tipo de analizador sintáctico: Descendente recursivo, LL(k).

Plataforma: Multiplataforma.

Se puede usar con: JFlex.

Licencia: Software libre. (Licencia BSD)

Entrada: Para introducir una gramática en ANTLR, se debe utilizar la notación EBNF y un conjunto de construcciones auxiliares. Permite especificar cada analizador (léxico, sintáctico, semántico) en un fuente independiente o en un único fuente. En este primer ejemplo optaremos por utilizar un solo fuente.

### CookCC

Analizadores generados: léxico y sintáctico.

Código generado:[Java](http://en.wikipedia.org/wiki/Java_(programming_language)), Plain, XML, Yacc.

Tipo de analizador sintáctico: LALR(1).

Plataforma: Multiplataforma.

Licencia: Software libre. (Licencia BSD)

Entrada: Se puede especificar el lexer/parser directamente en código Java. También puede ser un fichero XML que contiene una sección para el lexer y otra para el parser (solamente una de ellas es necesaria). También se permiten ficheros de entrada tradicionales de yacc/bison con algunas restricciones.

### Grammatica

Analizadores generados: sintáctico.

Código generado: [C](http://en.wikipedia.org/wiki/C_(programming_language))#, Java

Tipo de analizador sintáctico: Descendente, LL(k).

Plataforma: Multiplataforma (debe ejecutarse en una plataforma que soporte JRE/JDK 1.1 ó superior).

Se puede usar con: JFlex

Licencia: Software libre (licencia LGPL).

Entrada: Gramática basada en una notación similar a BNF.

Mejora herramientas similares (como Yacc y ANTLR) por la creación de código fuente bien comentado y legible, por tener la recuperación automática de errores y mensajes de error detallados, así como soporte para probar y depurar las gramáticas sin generar código fuente

### JavaCC

Analizadores generados: léxico y sintáctico.

Código generado:[Java](http://en.wikipedia.org/wiki/Java_(programming_language)).

Tipo de analizador sintáctico: Descendente recursivo, LL(k).

Plataforma: Multiplataforma.

Se puede usar con:

Licencia: Software libre. (Licencia BSD)

Entrada: Para introducir una gramática se debe utilizar la notación EBNF.

### YaCC

Analizadores generados: sintáctico.

Código generado: [C, C++.](http://en.wikipedia.org/wiki/C_(programming_language))

Tipo de analizador sintáctico: Ascendente, LALR(1).

Plataforma: Unix.

Se puede usar con: Requiere un analizador léxico como Lex, Flex.

Licencia: Open Source.

Entrada: Gramática basada en una notación similar a BNF.

## Comparativa de generadores léxicos y sintácticos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Nombre** | **Algoritmo** | **Notacion Gramatical** | **Entrada** | **Salida** | **Lexico** |
| ANTLR | Descendente | EBNF | Mixto | C, C++, Java, Ada, Perl,Python, Ruby, otros | generado |
| Yacc | LALR(1) | YACC | Mixto | C | Externo |
| JavaCC | Descendente (k) | EBNF | Mixto | Java | generado |
| JFLAP | Descendente(1), LALR(1) | ¿? | ¿? | Java | ¿? |
| Grammatica | Descendente(k) | dialecto BNF | Por separado | C#; Java | generado |
| CookCC | LALR(1) | Java, XML, Yacc | Mixto | Java | generado |

## Conclusiones

Tras el estudio y la comparativa de las diferentes herramientas utilizadas para la generación de analizadores léxicos y sintácticos, nuestra herramienta elegida por el momento como mejor opción es JavaCC.

No nos parecen buenas opciones las herramientas Lex, Flex, Yacc, dado que generan código en C, C++ y dicho lenguaje no está muy dominado por todos los miembros del grupo.

JFlex, JLex y Ragel son descartadas dado que estas herramientas generan solamente analizadores léxicos y por mayor comodidad creemos que es más conveniente utilizar una herramienta que genere también el analizador sintáctico. Lo mismo se aplica a Grammatica ya que genera solamente analizadores sintácticos.

Al no tener suficiente información sobre sus características y funcionamiento, descartamos la aplicación JFlap.

CookCC no incluye tanta documentación como JavaCC y ANTLR por lo cual sería más complicado aprender al usarla.

Se supone que es la herramienta más utilizada en Java. Los propietarios estiman en cientos de miles el número de descargas de la herramienta y los foros de discusión congregan a miles de usuarios interesados en JavaCC.

Se trata de una herramienta desarrollada en Java, lenguaje de programación ampliamente conocido por todos los miembros del grupo. Además, dispone de una amplia documentación, facilitando el uso de la misma.

JavaCC integra en una misma herramienta al analizador lexicográfico y al sintáctico, y el código que genera es independiente de cualquier biblioteca externa, lo que le confiere independencia respecto al entorno. Además, ofrece muchas opciones diferentes para personalizar su comportamiento y el comportamiento de los analizadores generados. Estos aspectos nos interesan ya que no tenemos un objetivo claro, al no conocer la gramática que vamos a utilizar. Dicha característica no se contempla en muchos de los generadores ?

Genera analizadores descendentes, permitiendo el uso de gramáticas de propósito general. Por defecto analiza gramáticas de tipo LL(1), pero permite fijar un tamaño de ventana de tokens mayor (para analizar gramáticas LL(k)) e incluso utilizar un tamaño adaptativo.

Otra de sus ventajas es que las especificaciones léxica (basada en expresiones regulares) y sintáctica (basada en el formato BNF) de la gramática que se va a analizar se incluyen en un mismo fichero. De esta forma la gramática puede ser leída y mantenida más fácilmente. Además, incluye la herramienta JJDoc que convierte los archivos de la gramática en archivos de documentación.

También incluye la herramienta JJTree, un preprocesador para el desarrollo de árboles con características muy potentes.

Es altamente eficiente, lo que lo hace apto para entornos profesionales y lo ha convertido en uno de los metacompiladores más extendidos (quizás el que más, por encima de JFlex/Cup).

De entre los generadores de analizadores sintácticos descendentes, JavaCC es uno de los que poseen mejor gestión de errores. Los analizadores generados por JavaCC son capaces, mediante las excepciones, de localizar exactamente la ubicación de los errores, proporcionando información diagnóstica completa.

# Diseño del analizador léxico, del gestor de errores y de la tabla de símbolos.

## 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) |
| IDENTIFICADOR | (IDENTIFICADOR,) |
| LIT\_CARACTER | (LIT\_CARACTER, [caracteres]) |
| LIT\_CADENA | (LIT\_CADENA, [índice tabla]) |
| PAL\_RESERVADA | (PAL\_RESERVADA, [índice 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) |

**IDENTIFICADOR:** secuencia de caracteres que nombran entidades.

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

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

**PAL\_RESERVADA:** secuencia de caracteres que nombran entidades.

**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 ;

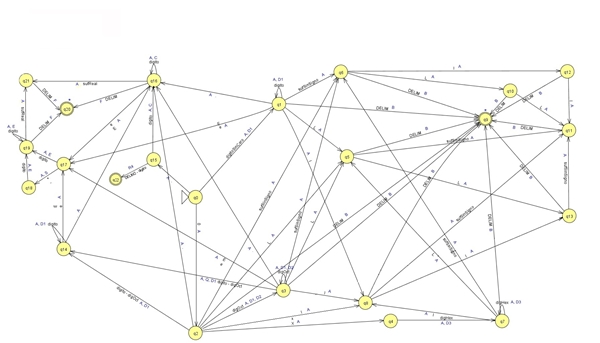
digSinCero = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 ;

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;

}

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 del analizador léxico y de los prototipos de la tabla de símbolos y del gestor de errores.

## Consideraciones previas a la implementación

Tras el estudio de las herramientas que generaban los analizadores, nuestra primera apuesta fue por usar JavaCC en la implementación de nuestro traductor.

En las primeras reuniones del grupo para la implementación del léxico, estuvimos deliberando si realmente, usar una herramienta para la generación del analizador léxico nos supondría un avance.

La principal desventaja del uso de una herramienta con respecto a la implementación manual de las acciones semánticas es la ausencia por parte de los componentes del grupo de conocimiento posterior sobre cómo funciona realmente un analizador léxico.

Lo expuesto anteriormente, junto con los conocimientos avanzados de todos los componentes del grupo con respecto a la programación en java, con muchos años de experiencia tanto en la universidad como en el ámbito laboral, nos han hecho decantarnos por la implementación manual de las acciones semánticas.

## Implementación analizador léxico

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

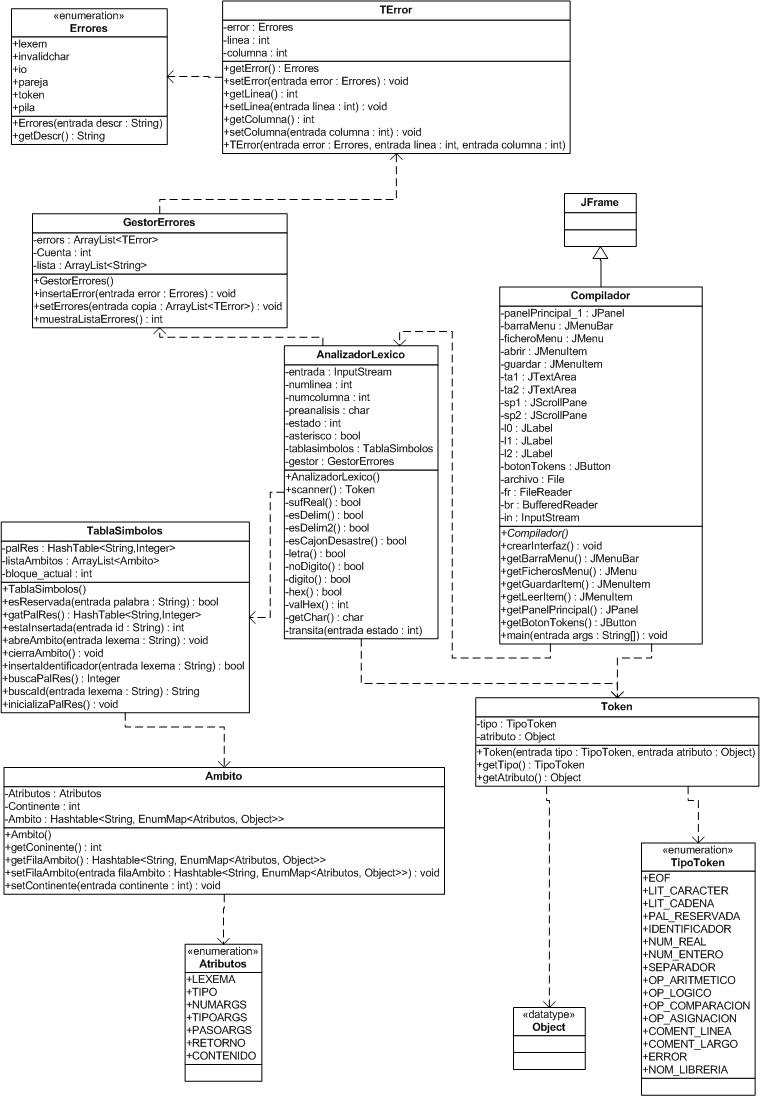
Se ha implementado una interfaz gráfica simple, que muestra dos áreas de texto. El área de la izquierda permite poder cargar un fichero guardado en el computador, o escribir directamente sobre el mismo pudiéndose guardar posteriormente lo escrito. También muestra un botón que lista los tokens generados por el analizador léxico en el área de texto de la derecha.

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.

Los tokens correspondientes a los números, los generamos, usando como valor del par, la correspondencia del mismo en base 10.

### 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).

## 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.

# Glosario

ATN: Augmented Transaction Network

LALR: Look-Ahead Left to Right Parser

LL: Left to right, Leftmost derivation

EBNF: Extended Backus–Naur Form

BNF: Backus-Naur form

DFA: Deterministic-finite automaton

CDDL = Licencia común de Desarrollo y Distribución

Mixto = Tabla de ER’s y fragmentos de código

LGPL = Licencia GNU GPL con una excepción de linkado

# Bibliografía

* <http://www.antlr.org/>
* <http://www.escet.urjc.es/~procesal/analizadores.html>
* <http://es.wikipedia.org/wiki/JavaCC>
* <http://javacc.java.net/>
* <http://www.lcc.uma.es/~galvez/ftp/libros/Compiladores.pdf>
* <http://en.wikipedia.org/wiki/Comparison_of_parser_generators>
* <http://www.cs.princeton.edu/~appel/modern/java/JLex/>
* <http://dinosaur.compilertools.net/>
* <http://www.gnu.org/software/flex/>
* <http://jflex.de/>