Manual de la gramática de JFlex y Cup

JFlex

Opciones y macros

- **%class Lexer** : Esto hace que la clase generada el nombre Lexer y que escriba el código en un archivo Lexer.java.
- **%cupsym Sym** : Esto hace que la clase generada de los simbolos se llame Sym.java
- **%public** : Hace que la clase generada sea pública (ya que la clase solo es accesible en su propio paquete de forma predeterminada).
- **%unicode** : Esta opcion hace que el escáner generado use el conjunto completo de caracteres de entrada Unicode, incluidos los puntos de código suplementarios: 0-0x10FFFF. % unicode no significa que el escáner leerá dos bytes a la vez. Lo que se lee y lo que constituye un personaje depende de la plataforma de tiempo de ejecución.
- **%line** : Activa la línea contando. La variable de miembro int yyline contiene el número de líneas (comenzando con 0) desde el comienzo de la entrada hasta el comienzo del token actual.
- **%column** : Activa la columna contando. La variable de miembro int yycolumn contiene el número de caracteres (comenzando con 0) desde el comienzo de la línea actual hasta el comienzo del token actual.
- **%cup** : cambia al modo de compatibilidad CUP para interactuar con un analizador CUP generado.
- **%char** : Activa el carácter contando. La variable miembro larga yychar contiene el número de caracteres (comenzando con 0) desde el comienzo de la entrada hasta el comienzo del token actual.
- **%ignorecase** : Esta opción hace que JFlex maneje todos los caracteres y cadenas en la especificación como si estuvieran especificados en mayúsculas y minúsculas. Esto permite una manera fácil de especificar un escáner para un idioma con palabras clave que no distinguen entre mayúsculas y minúsculas.

Codigo de usuario

En esta seccion se muestra el codigo establecido por el usuario, en donde se muestra las variables usadas, los metodos para crear los simbolos y el metodo para guardar los errores.

```
%{
    /**
     * Guarda el texto de las cadenas
     */
    private StringBuilder string;

    /**
```

```
     * Creador de simbolos complejos
     */
    private ComplexSymbolFactory symbolFactory;

    /**
     * analizador
     */
    private AritLanguage aritLanguage;

    /**
     * Constructor del analizador lexico
     *
     * @param in Entrada que se va analizar
     * @param symbolFactory creador de simbolos complejos
     */
    public Lexer(java.io.Reader in, ComplexSymbolFactory symbolFactory, AritLanguage
aritLanguage) {
        this(in);
        string = new StringBuilder();
        this.symbolFactory = symbolFactory;
        this.aritLanguage = aritLanguage;
    }

    /**
     * Metodo que devuelve un nuevo java_cup.runtime.Symbol
     *
     * @param name nombre que recibira el simbolo
     * @param sym numero de token
     * @param value valor que recibira el simbolo
     * @param buflength tam del valor
     * @return java_cup.runtime.Symbol
     */
    private Symbol symbol(String name, int sym, Object value, int buflength) {
        Location left = new Location(yyline + 1, yycolumn + yylength() - buflength,
yychar + yylength() - buflength);
        Location right= new Location(yyline + 1, yycolumn + yylength(), yychar +
yylength());
        return symbolFactory.newSymbol(name, sym, left, right, value);
    }

    /**
     * Metodo que devuelve un nuevo java_cup.runtime.Symbol
     *
     * @param name nombre que recibira el simbolo
     * @param sym numero de token
     * @return java_cup.runtime.Symbol
     */
    private Symbol symbol(String name, int sym) {
        Location left = new Location(yyline + 1, yycolumn + 1, yychar);
        Location right= new Location(yyline + 1, yycolumn + yylength(), yychar +
```

```
yylength());
        return symbolFactory.newSymbol(name, sym, left, right);
    }

    /**
     * Devuelve un nuevo java_cup.runtime.Symbol
     *
     * @param name nombre que recibira el simbolo
     * @param sym numero de token
     * @param val valor que recibira el simbolo
     * @return java_cup.runtime.Symbol
     */
    private Symbol symbol(String name, int sym, Object val) {
        Location left = new Location(yyline + 1, yycolumn + 1, yychar);
        Location right= new Location(yyline + 1, yycolumn + yylength(), yychar +
yylength());
        return symbolFactory.newSymbol(name, sym, left, right, val);
    }

    /**
     * Guarda los errores en el manejador
     *
     * @param message mensaje del error
     */
    private void error(String message) {
        aritLanguage.addLexicalError(message, new NodeInfo(yyline + 1, yycolumn + 1,
aritLanguage.filename));
    }
%}

%eofval{
    return symbolFactory.newSymbol(
        "EOF", Sym.EOF,
        new Location(yyline + 1, yycolumn + 1, yychar),
        new Location(yyline + 1, yycolumn + 1, yychar + 1)
    );
%eofval}
```

Patrones

Patrones lexicos usados en el paso de reglas y acciones

```
Digito          = [0-9]
Letra           = [a-zA-Z]

/**
 *  Finalización de línea:
 *
```

```
 *  Carácter salto de línea (LF ASCII)
 *  Carácter retorno de carro (CR ASCII)
 *  Carácter retorno de carro (CR ASCII) seguido de carácter salto de línea (LF
ASCII)
 */
Fin_linea        = \r|\n|\r\n

/**
 *  Espacios en blanco:
 *
 *  Espacio (SP ASCII)
 *  Tabulación horizontal (HT ASCII)
 *  Caracteres de finalización de línea
 */
Espacios        = {Fin_linea} | [ \t\f]

/* Identificadores */
Identificador   = ((["."]({Letra}|["_"]|["."]))|{Letra})({Letra}|{Digito}|["_"]|
["."])*

/* literales */
Integer_Literal = {Digito}+
Decimal_Literal = {Digito}+["."]{Digito}+
Boolean_Literal = true|false
Null_Literal    = null

/* Estados */

/**
 *  Comentarios simples:
 *
 *  Los comentarios de una línea que serán delimitados al inicio con el símbolo #
 *  y al final con un carácter de finalización de línea.
 */
%state COMENTARIO_DE_FIN_DE_LINEA

/**
 *  Comentarios múltiples:
 *
 *  Los comentarios múltiples serán delimitados al inicio con los símbolos #*
 *  y al final con un los símbolos *#
 */
%state COMENTARIO_TRADICIONAL

/**
 * string: "<Caracteres ASCII> "
 */
%state CADENA
```

Reglas y acciones

Estado inicial

En este estado se define las palabras claves del lenguaje, las literales y su conversion, los identificadores, separadores, espacios en blanco y la llamada de los estados de los comentarios y cadena.

```
<YYINITIAL> {
    /*
        palabras claves del lenguaje
        se llama al metodo al metodo symbol(nombre, token)
    */
    "continue"          { return symbol("`continue`", Sym.CONTINUE); }
    "function"          { return symbol("`function`", Sym.FUNCTION); }
    "default"           { return symbol("`default`", Sym.DEFAULT); }
    "return"            { return symbol("`return`", Sym.RETURN); }
    "switch"            { return symbol("`switch`", Sym.SWITCH); }
    "break"             { return symbol("`break`", Sym.BREAK); }
    "while"             { return symbol("`while`", Sym.WHILE); }
    "case"              { return symbol("`case`", Sym.CASE); }
    "else"              { return symbol("`else`", Sym.ELSE); }
    "for"               { return symbol("`for`", Sym.FOR); }
    "in"                { return symbol("`in`", Sym.IN); }
    "do"                { return symbol("`do`", Sym.DO); }
    "if"                { return symbol("`if`", Sym.IF); }


    /* literales */
    {Integer_Literal}   {
                            int valor = 0;
                            try {
                                valor = Integer.parseInt(yytext());
                            } catch (NumberFormatException ex) {
                                error("Error: Numero "+ yytext () +" fuera del rango
de un integer.");
                            }
                            return symbol("`Integer Literal`", Sym.LIT_ENTERO,
valor);
                        }

    {Decimal_Literal}   {
                            double valor = 0;
                            try {
                                valor = Double.parseDouble(yytext());
                            } catch (NumberFormatException ex) {
                                error("Error: "+ yytext () +" fuera del rango de un
decimal.");
                            }
```

```
                            return symbol("`Decimal Literal`", Sym.LIT_DECIMAL,
valor);
                        }

    {Boolean_Literal}   { return symbol(yytext(), Sym.LIT_BOOLEANO,
Boolean.parseBoolean(yytext())); }

    {Null_Literal}      { return symbol("null", Sym.NULL); }

    \"                  { string.setLength(0); yybegin(CADENA); }

    /* nombres */
    {Identificador}     { return symbol("`Identifier: '" + yytext() + "'`", Sym.ID,
yytext().toLowerCase()); }

    /* separadores */
    "=>"                { return symbol("`=>`", Sym.LAMBDA);              }
    "=="                { return symbol("`==`", Sym.IGUAL_QUE);           }
    "!="                { return symbol("`!=`", Sym.DIFERENTE_QUE);       }
    ">="                { return symbol("`>=`", Sym.MAYOR_IGUAL_QUE);     }
    "<="                { return symbol("`<=`", Sym.MENOR_IGUAL_QUE);     }
    "%%"                { return symbol("`%`",  Sym.MODULO);              }
    "+"                 { return symbol("`+`",  Sym.MAS);                 }
    "-"                 { return symbol("`-`",  Sym.MENOS);               }
    "*"                 { return symbol("`*`",  Sym.MULT);                }
    "/"                 { return symbol("`/`",  Sym.DIV);                 }
    "^"                 { return symbol("`^`",  Sym.POTENCIA);            }
    "="                 { return symbol("`=`",  Sym.IGUAL);               }
    ">"                 { return symbol("`>`",  Sym.MAYOR_QUE);           }
    "<"                 { return symbol("`<`",  Sym.MENOR_QUE);           }
    "?"                 { return symbol("`?`",  Sym.INTERROGANTE);        }
    ":"                 { return symbol("`:`",  Sym.DOS_PUNTOS);          }
    "|"                 { return symbol("`|`",  Sym.OR);                  }
    "&"                 { return symbol("`&`",  Sym.AND);                 }
    "!"                 { return symbol("`!`",  Sym.NOT);                 }
    "("                 { return symbol("`(`",  Sym.PAR_IZQ);             }
    ")"                 { return symbol("`)`",  Sym.PAR_DER);             }
    "["                 { return symbol("`[`",  Sym.COR_IZQ);             }
    "]"                 { return symbol("`]`",  Sym.COR_DER);             }
    ";"                 { return symbol("`;`",  Sym.PUNTO_COMA);          }
    ","                 { return symbol("`,`",  Sym.COMA);                }
    "{"                 { return symbol("`{`",  Sym.LLAVE_IZQ);           }
    "}"                 { return symbol("`}`",  Sym.LLAVE_DER);           }

    /* espacios en blanco */
    {Espacios}          { /* IGNORAR ESPACIOS */ }

    /* comentarios */
    "#*"                { yybegin(COMENTARIO_TRADICIONAL); }
```

```
    "#"                 { yybegin(COMENTARIO_DE_FIN_DE_LINEA); }
}
```

Estado de comentario tradicional

Los comentarios múltiples serán delimitados al inicio con los símbolos #_ y al final con un los símbolos _#

```
<COMENTARIO_TRADICIONAL> {
    "*#"                { yybegin(YYINITIAL); }
    <<EOF>>             { yybegin(YYINITIAL); }
    [^]                 { /* IGNORAR CUALQUIER COSA */ }
}
```

Estado de comentario de fin de linea

Los comentarios de una línea que serán delimitados al inicio con el símbolo # y al final con un carácter de finalización de línea.

```
<COMENTARIO_DE_FIN_DE_LINEA> {
    {Fin_linea}             { yybegin(YYINITIAL); }
    .                       { /* IGNORAR */  }
}
```

Estado de cadena

En este estado se va almacenando el cuerpo de la cadena en la variable string.

```
<CADENA> {
    /* Fin de cadena */
    \"                  {
                            yybegin(YYINITIAL);
                            return symbol("`String Literal`", Sym.LIT_STRING,
string.toString(), string.length());
                        }

    /* Secuencias de escape */
    "\\\""              { string.append('\"'); }
    "\\\\"              { string.append('\\'); }
    "\\n"               { string.append('\n'); }
    "\\r"               { string.append('\r'); }
    "\\t"               { string.append('\t'); }
    \\.                 { error("Error: no se esperaba el escape \\." + yytext());
```

```
string.append(yytext()); }

    {Fin_linea}         {
                            yybegin(YYINITIAL);
                            error("Error: final de linea inesperado.");
                        }

    [^\r\n\"\\]+        { string.append(yytext()); }
}
```

Errores

En esta regla se guarda cualquier error encontrado durante el analisis.

```
/* Cualquier cosa que no coincida con lo de arriba es un error. */
[^] {
    error("Error: Carácter no valido '"+ yytext () +"'.");
}
```

CUP

Codigo usuario

```
 /**
     * Manejador de tipos
     */
    private static final TypeFacade TYPE_FACADE = TypeFacade.getInstance();

    /**
     * analizador
     */
    private AritLanguage aritLanguage;

    /**
     * Nombre del archivo que esta analizando
     */
    private String filename;

    /**
     * Constructor del analizador sintactico
     *
     * @param scann  Analizador lexico
     * @param sf     Fabrica de simbolos
     */
    public Parser(Lexer scann, ComplexSymbolFactory sf, AritLanguage aritLanguage) {
```

```
        super(scann, sf);
        this.aritLanguage = aritLanguage;
        this.filename = aritLanguage.filename;
    }

    /**
     *
     * Método al que se llama automaticamente ante algun error sintactico.
     *
     * @param s simbolo que provoco el error
     */
    @Override
    public void syntax_error(java_cup.runtime.Symbol s) {
        ComplexSymbol cs = (ComplexSymbol) s;
        error("Error: No se esperaba el siguiente simbolo " + cs.getName() + ".",
            cs.getLeft().getLine(), cs.getRight().getColumn());
    }

    /**
     *
     * Método al que se llama en el momento en que ya no es posible una
     * recuperacion de errores.

     * @param s simbolo que provoco el error
     * @throws Exception
     */
    @Override
    public void unrecovered_syntax_error(java_cup.runtime.Symbol s) throws Exception
{
        ComplexSymbol cs = (ComplexSymbol) s;
        error("Error irrecuperable provocado simbolo " + cs.getName() + ".",
        cs.getLeft().getLine(), cs.getRight().getColumn());
    }

    /**
     * Guarda los errores en el manejador
     *
     * @param message mensaje de por que se provoco el error
     * @param line linea donde se encuentra el error
     * @param column columna donde se encuentra el error
     */
    private void error(String message, int line, int column) {
        List<Integer> ids = this.expected_token_ids();
        LinkedList<String> list = new LinkedList<>();
        for (Integer expected : ids) {
            list.add(this.symbl_name_from_id(expected));
        }
        if (list.isEmpty()) {
            aritLanguage.addSyntacticError(message, new NodeInfo(line, column,
filename));
```

```
        } else {
            aritLanguage.addSyntacticError(message + "\nSi no que se esperaba alguno
de los siguientes tokens: " + list,
                    new NodeInfo(line, column, filename));
        }
    }
```

Terminales

Terminales (tokens devueltos por el escáner).

```
/* Palabras claves */
terminal CONTINUE, FUNCTION, DEFAULT, RETURN, SWITCH, BREAK, WHILE, CASE, ELSE, FOR,
IN, DO, IF;

/* Literales */
terminal Integer LIT_ENTERO;
terminal Double  LIT_DECIMAL;
terminal Boolean LIT_BOOLEANO;
terminal String  LIT_STRING;
terminal String  ID;
terminal String  NULL;

/* Separadores */
terminal LAMBDA, IGUAL_QUE, DIFERENTE_QUE, MAYOR_IGUAL_QUE, MENOR_IGUAL_QUE, MODULO,
MAS, MENOS, MULT, DIV, POTENCIA, IGUAL, MAYOR_QUE, MENOR_QUE, INTERROGANTE,
DOS_PUNTOS, OR, AND, NOT, PAR_IZQ, PAR_DER, COR_IZQ, COR_DER, PUNTO_COMA,
COMA, LLAVE_IZQ, LLAVE_DER, UMENOS;
```

No Terminales

Los no terminales son usados en las producciones de la gramatica.

```
/* No Terminales */
non terminal compilation_unit;
non terminal ArrayList<AstNode> global_statements;
non terminal AstNode global_statement;
non terminal Function function_declaration;
non terminal ArrayList<FormalParameter> formal_parameter_list;
non terminal FormalParameter formal_parameter;
non terminal Block block;
non terminal ArrayList<AstNode> block_statements;
non terminal AstNode statement;
non terminal Break break_statement;
non terminal Continue continue_statement;
```

```
non terminal Return return_statement;
non terminal IfStatement if_statement;
non terminal ArrayList<SubIf> if_list;
non terminal SwitchStm switch_statement;
non terminal ArrayList<CaseStm> switch_labels;
non terminal CaseStm switch_label;
non terminal WhileStm while_statement;
non terminal DoWhileStm do_while_statement;
non terminal ForStm for_statement;
non terminal Expression statement_expression;
non terminal Expression expression;
non terminal Expression var_assigment;
non terminal ArrayList<Access> access_list;
non terminal Access access;
non terminal Expression function_call;
non terminal ArrayList<Expression> argument_list;
non terminal Expression default_exp;
```

Precedencia utilizada

```
                            /* Precedencias */

/*Asociatividad */ /* Operador */                                          /* Nivel
*/
precedence right    IGUAL;                                                 // 1
precedence right    INTERROGANTE, DOS_PUNTOS;                              // 2
precedence left     OR;                                                    // 3
precedence left     AND;                                                   // 4
precedence left     DIFERENTE_QUE, IGUAL_QUE;                              // 5
precedence nonassoc MAYOR_IGUAL_QUE, MAYOR_QUE, MENOR_IGUAL_QUE, MENOR_QUE; // 6
precedence left     MAS, MENOS;                                            // 7
precedence left     MULT, DIV, MODULO;                                     // 8
precedence left     POTENCIA;                                              // 9
precedence right    UMENOS, NOT;                                           // 10;
precedence left     COR_IZQ, COR_DER, PAR_IZQ, PAR_DER;                    // 11;
```

Reglas gramaticales

El no terminal inicial es compilation_unit.

```
start with compilation_unit;
```

compilation_unit

Este no terminal devuelve el AST.

```
/*
    compilation_unit -> global_statements
 */
compilation_unit
    ::= global_statements:astNodes {:
        if (astNodes != null) aritLanguage.setAstNodes(astNodes);
    :}
;
```

global_statements

Este no terminal produce una lista de sentencias.

```
/*
    global_statements -> global_statements global_statement
                    |    global_statement
 */
global_statements
    ::= global_statements:statements global_statement:statement {:
            RESULT = statements;
            if (statement != null) RESULT.add(statement);
        :}
    |   global_statement:statement {:
            RESULT = new ArrayList<>();
            if (statement != null) RESULT.add(statement);
        :}
;
```

global_statement

Este no terminal devuelve una sentencia valida en el ambito global.

```
/*
    global_statement -> statement
                    |   function_declaration [;]
*/
global_statement
    ::= statement:statement {: RESULT = statement; :}
    |   function_declaration:function {: RESULT = function; :}
    |   function_declaration:function PUNTO_COMA {: RESULT = function; :}
;
```

function_declaration

Este no terminal devuelve la declaracion de una funcion.

```
/*
    function_declaration -> id = 'function' '(' [formal_parameter_list] ')' block
                          |   id = '(' [formal_parameter_list] ')' '=>' block
*/
function_declaration
    ::= ID:id IGUAL FUNCTION PAR_IZQ formal_parameter_list:parameters PAR_DER
block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new Function(info, id, parameters, block);
        :}
    |   ID:id IGUAL FUNCTION PAR_IZQ PAR_DER block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new Function(info, id, block);
        :}
    |   ID:id IGUAL PAR_IZQ ID:idParameter COMA formal_parameter_list:parameters
PAR_DER LAMBDA block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            NodeInfo infoParameter = new NodeInfo(idParameterxleft.getLine(),
idParameterxright.getColumn(), filename);
            parameters.add(0, new FormalParameter(infoParameter, idParameter));
            RESULT = new Function(info, id, parameters, block);
        :}
    |   ID:id IGUAL PAR_IZQ ID:idParameter IGUAL expression:expParameter COMA
formal_parameter_list:parameters PAR_DER LAMBDA block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            NodeInfo infoParameter = new NodeInfo(idParameterxleft.getLine(),
idParameterxright.getColumn(), filename);
            parameters.add(0, new FormalParameter(infoParameter, idParameter,
expParameter));
            RESULT = new Function(info, id, parameters, block);
        :}
    |   ID:id IGUAL PAR_IZQ ID:idParameter PAR_DER LAMBDA block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            ArrayList<FormalParameter> parameters = new ArrayList<>();
            NodeInfo infoParameter = new NodeInfo(idParameterxleft.getLine(),
idParameterxright.getColumn(), filename);
            parameters.add(new FormalParameter(infoParameter, idParameter));
            RESULT = new Function(info, id, parameters, block);
        :}
    |   ID:id IGUAL PAR_IZQ ID:idParameter IGUAL expression:expParameter PAR_DER
```

```
LAMBDA block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            ArrayList<FormalParameter> parameters = new ArrayList<>();
            NodeInfo infoParameter = new NodeInfo(idParameterxleft.getLine(),
idParameterxright.getColumn(), filename);
            parameters.add(new FormalParameter(infoParameter, idParameter,
expParameter));
            RESULT = new Function(info, id, parameters, block);
        :}
    |   ID:id IGUAL PAR_IZQ PAR_DER LAMBDA block:block {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new Function(info, id, block);
        :}
;
```

formal_parameter_list

Devuelve una lista de parametros para una funcion.

```
/*
    formal_parameter_list -> formal_parameter_list ',' formal_parameter
                        |    formal_parameter
*/
formal_parameter_list
    ::= formal_parameter_list:parameters COMA formal_parameter:parameter {: RESULT =
parameters; RESULT.add(parameter); :}
    |   formal_parameter:parameter {: RESULT = new ArrayList<>();
RESULT.add(parameter); :}
;
```

formal_parameter

Devuelve un parametro de una funcion.

```
/*
    formal_parameter -> id ['=' expression]
*/
formal_parameter
    ::= ID:id {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new FormalParameter(info, id);
        :}
```

```
    |   ID:id IGUAL expression:exp {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new FormalParameter(info, id, exp);
        :}
;
```

block

Devuelve una lista de sentencias.

```
/*
    block -> '{' [block_statements] '}'
*/
block
    ::= LLAVE_IZQ:llave LLAVE_DER {:
            NodeInfo info = new NodeInfo(llavexleft.getLine(),
llavexright.getColumn(), filename);
            RESULT = new Block(info);
        :}
    |   LLAVE_IZQ:llave block_statements:statements LLAVE_DER {:
            NodeInfo info = new NodeInfo(llavexleft.getLine(),
llavexright.getColumn(), filename);
            RESULT = new Block(info, statements);
        :}
;
```

block_statements

Devuelve una lista de sentencias.

```
/*
    block_statements -> block_statements statement
                    |   statement
*/
block_statements
    ::= block_statements:statements statement:statement {:
            RESULT = statements;
            if (statement != null) RESULT.add(statement);
        :}
    |   statement:statement {:
            RESULT = new ArrayList<>();
            if (statement != null) RESULT.add(statement);
        :}
;
```

break_statement

Devuelve una sentencia break.

```
/* break_statement -> 'break' [';'] */
break_statement
    ::= BREAK:stm {: RESULT = new Break(new NodeInfo(stmxleft.getLine(),
stmxright.getColumn(), filename)); :}
    |   BREAK:stm PUNTO_COMA {: RESULT = new Break(new NodeInfo(stmxleft.getLine(),
stmxright.getColumn(), filename)); :}
;
```

continue_statement

Devuelve una sentencia continue.

```
/* continue_statement -> 'continue' [';']*/
continue_statement
    ::= CONTINUE:stm {: RESULT = new Continue(new NodeInfo(stmxleft.getLine(),
stmxright.getColumn(), filename)); :}
    |   CONTINUE:stm PUNTO_COMA {: RESULT = new Continue(new
NodeInfo(stmxleft.getLine(), stmxright.getColumn(), filename)); :}
;
```

return_statement

Devuelve una sentencia return.

```
/* return_statement -> 'return' [ '(' expression ')'] [';'] */
return_statement
    ::= RETURN:r PUNTO_COMA {:
            NodeInfo info = new NodeInfo(rxleft.getLine(), rxleft.getColumn(),
filename);
            RESULT = new Return(info);
        :}
    |   RETURN:r {:
            NodeInfo info = new NodeInfo(rxleft.getLine(), rxleft.getColumn(),
filename);
            RESULT = new Return(info);
        :}
    |   RETURN:r PAR_IZQ expression:exp PAR_DER {:
            NodeInfo info = new NodeInfo(rxleft.getLine(), rxleft.getColumn(),
```

```
filename);
            RESULT = new Return(info, exp);
        :}
    |   RETURN:r PAR_IZQ expression:exp PAR_DER PUNTO_COMA {:
            NodeInfo info = new NodeInfo(rxleft.getLine(), rxleft.getColumn(),
filename);
            RESULT = new Return(info, exp);
        :}
;
```

if_statement

No terminal que retorna la estructura de un IF, esta compuesto por una lista de if's y la sentencia 'else' puede venir o no.

```
/* if_statement -> if_list [ 'else' block ] */
if_statement
    ::= if_list:list ELSE:else_ block:block {:
            NodeInfo info = new NodeInfo(else_xleft.getLine(),
else_xleft.getColumn(), filename);
            list.add(new SubIf(info, block));
            info = new NodeInfo(listxleft.getLine(), listxleft.getColumn(),
filename);
            RESULT = new IfStatement(info, list);
        :}
    |   if_list:list {:
            NodeInfo info = new NodeInfo(listxleft.getLine(), listxleft.getColumn(),
filename);
            RESULT = new IfStatement(info, list);
        :}
;
```

if_list

No terminal que devuelve una lista de if's.

```
/*
    if_list -> if_list 'else' 'if' '(' expression ')' block
            |  'if' '(' expression ')' block
*/
if_list
    ::= if_list:list ELSE IF:if_ PAR_IZQ expression:exp PAR_DER block:block {:
            RESULT = list;
            NodeInfo info = new NodeInfo(if_xleft.getLine(), if_xleft.getColumn(),
filename);
```

```
            RESULT.add(new SubIf(info, exp, block));
        :}
    |   IF:if_ PAR_IZQ expression:exp PAR_DER block:block {:
            NodeInfo info = new NodeInfo(if_xleft.getLine(), if_xleft.getColumn(),
filename);
            RESULT = new ArrayList<>();
            RESULT.add(new SubIf(info, exp, block));
        :}
;
```

switch_statement

No terminal que devuelve una sentencia switch.

```
switch_statement
    ::= SWITCH:stm PAR_IZQ expression:exp PAR_DER LLAVE_IZQ switch_labels:labels
LLAVE_DER {:
            NodeInfo info = new NodeInfo(stmxleft.getLine(), stmxright.getColumn(),
filename);
            RESULT = new SwitchStm(info, exp, labels);
        :}
    |   SWITCH:stm PAR_IZQ expression:exp PAR_DER LLAVE_IZQ LLAVE_DER {:
            NodeInfo info = new NodeInfo(stmxleft.getLine(), stmxright.getColumn(),
filename);
            RESULT = new SwitchStm(info, exp, null);
        :}
;
```

switch_labels

No terminal el cual devuelve una lista de etiquetas de un switch.

```
switch_labels
    ::= switch_labels:labels switch_label:label {:
            RESULT = labels;
            if (label != null) RESULT.add(label);
        :}
    |   switch_label:label {:
            RESULT = new ArrayList<>();
            if (label != null) RESULT.add(label);
        :}
;
```

switch_label

No terminal que devuelve una etiqueta.

```
switch_label
    ::= CASE:label expression:exp DOS_PUNTOS block_statements:block {:
            NodeInfo info = new NodeInfo(labelxleft.getLine(),
labelxright.getColumn(), filename);
            RESULT = new CaseStm(info, exp, block);
        :}
    |   CASE:label expression:exp DOS_PUNTOS {:
            NodeInfo info = new NodeInfo(labelxleft.getLine(),
labelxright.getColumn(), filename);
            RESULT = new CaseStm(info, exp, null);
        :}
    |   DEFAULT:label DOS_PUNTOS block_statements:block {:
            NodeInfo info = new NodeInfo(labelxleft.getLine(),
labelxright.getColumn(), filename);
            RESULT = new CaseStm(info, null, block);
        :}
    |   DEFAULT:label DOS_PUNTOS {:
            NodeInfo info = new NodeInfo(labelxleft.getLine(),
labelxright.getColumn(), filename);
            RESULT = new CaseStm(info, null, null);
        :}
;
```

while_statement

No terminal que devuelve una sentencia while.

```
while_statement
    ::= WHILE:stm PAR_IZQ expression:exp PAR_DER block:block  {:
            NodeInfo info = new NodeInfo(stmxleft.getLine(), stmxright.getColumn(),
filename);
            RESULT = new WhileStm(info, exp, block);
        :}
;
```

do_while_statement

No terminal que devuelve una sentencia do while.

```
do_while_statement
    ::= DO:stm block:block WHILE PAR_IZQ expression:exp PAR_DER {:
            NodeInfo info = new NodeInfo(stmxleft.getLine(), stmxright.getColumn(),
```

```
filename);
            RESULT = new DoWhileStm(info, exp, block);
        :}
    |   DO:stm block:block WHILE PAR_IZQ expression:exp PAR_DER PUNTO_COMA {:
            NodeInfo info = new NodeInfo(stmxleft.getLine(), stmxright.getColumn(),
filename);
            RESULT = new DoWhileStm(info, exp, block);
        :}
;
```

for_statement

No terminal que devuelve una sentencia for.

```
for_statement
    ::= FOR:stm PAR_IZQ ID:id IN expression:exp PAR_DER block:block {:
            NodeInfo info = new NodeInfo(stmxleft.getLine(), stmxright.getColumn(),
filename);
            RESULT = new ForStm(info, id, exp, block);
        :}
;
```

statement

Sentencias validas en cualquier ambito.

```
statement
    ::= statement_expression:expression PUNTO_COMA {: RESULT = expression; :}
    |   statement_expression:expression {: RESULT = expression; :}
    |   break_statement:statement {: RESULT = statement; :}
    |   continue_statement:statement {: RESULT = statement; :}
    |   return_statement:statement {: RESULT = statement; :}
    |   if_statement:statement {: RESULT = statement; :}
    |   if_statement:statement PUNTO_COMA {: RESULT = statement; :}
    |   switch_statement:statement {: RESULT = statement; :}
    |   switch_statement:statement PUNTO_COMA {: RESULT = statement; :}
    |   while_statement:statement {: RESULT = statement; :}
    |   while_statement:statement PUNTO_COMA {: RESULT = statement; :}
    |   do_while_statement:statement {: RESULT = statement; :}
    |   for_statement:statement {: RESULT = statement; :}
    |   for_statement:statement PUNTO_COMA {: RESULT = statement; :}
    |   error PUNTO_COMA
;
```

statement_expression

Expresiones que tambien son sentencias.

```
statement_expression
    ::= var_assigment:assigment {: RESULT = assigment; :}
    |   function_call:functionCall {: RESULT = functionCall; :}
;
```

var_assigment

Asignacion a una variable normal.

```
var_assigment
    ::= ID:id IGUAL expression:expression {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new Assignment(info, id, expression);
        :}
    |   ID:id access_list:list IGUAL expression:expression {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new StructureAssignment(info, id, list, expression);
        :}
;
```

access_list

Lista de accesos a una estructura.

```
access_list
    ::= access_list:list access:access {: RESULT = list; if (access != null)
RESULT.add(access); :}
    |   access:access {: RESULT = new ArrayList<>(); if (access != null)
RESULT.add(access); :}
;
```

access

Acceso a una estructura

```
access
    ::= COR_IZQ expression:exp1 COMA expression:exp2 COR_DER {:
            NodeInfo info = new NodeInfo(exp1xleft.getLine(),
exp1xright.getColumn(), filename);
            RESULT = new Access(info, exp1, exp2);
        :}
    |   COR_IZQ expression:exp COMA COR_DER {:
            NodeInfo info = new NodeInfo(expxleft.getLine(), expxright.getColumn(),
filename);
            RESULT = new Access(info, exp, Access.Type.TWO_MATRIX);
        :}
    |   COR_IZQ COMA expression:exp COR_DER {:
            NodeInfo info = new NodeInfo(expxleft.getLine(), expxright.getColumn(),
filename);
            RESULT = new Access(info, exp, Access.Type.THREE_MATRIX);
        :}
    |   COR_IZQ expression:exp COR_DER {:
            NodeInfo info = new NodeInfo(expxleft.getLine(), expxright.getColumn(),
filename);
            RESULT = new Access(info, exp, Access.Type.NORMAL);
        :}
    |   COR_IZQ COR_IZQ expression:exp COR_DER COR_DER {:
            NodeInfo info = new NodeInfo(expxleft.getLine(), expxright.getColumn(),
filename);
            RESULT = new Access(info, exp, Access.Type.TWO_LIST);
        :}
;
```

function_call

Llamada de una funcion.

```
function_call
    ::= ID:id PAR_IZQ PAR_DER {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new FunctionCall(info, id);
        :}
    |   ID:id PAR_IZQ argument_list:arguments PAR_DER {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new FunctionCall(info, id, arguments);
        :}
;
```

expression

Gramatica ambigua de expresion, se utilizo las precedencias indicadas anteriormente.

```
expression
    ::= function_call:expression {: RESULT = expression; :}
    |   expression:condition INTERROGANTE:op expression:exp1 DOS_PUNTOS
expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Ternary(info, condition, exp1, exp2);
        :}
    |   expression:exp1 OR:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Logical(info, exp1, exp2, Logical.Operator.OR);
        :}
    |   expression:exp1 AND:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Logical(info, exp1, exp2, Logical.Operator.AND);
        :}
    |   expression:exp1 DIFERENTE_QUE:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Comparator(info, exp1, exp2, Comparator.Operator.UNEQUAL);
        :}
    |   expression:exp1 IGUAL_QUE:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Comparator(info, exp1, exp2, Comparator.Operator.EQUAL);
        :}
    |   expression:exp1 MAYOR_IGUAL_QUE:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Comparator(info, exp1, exp2,
Comparator.Operator.GREATER_THAN_OR_EQUAL_TO);
        :}
    |   expression:exp1 MAYOR_QUE:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Comparator(info, exp1, exp2,
Comparator.Operator.GREATER_THAN);
        :}
    |   expression:exp1 MENOR_IGUAL_QUE:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Comparator(info, exp1, exp2,
Comparator.Operator.LESS_THAN_OR_EQUAL_TO);
```

```
        :}
    |   expression:exp1 MENOR_QUE:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Comparator(info, exp1, exp2,
Comparator.Operator.LESS_THAN);
        :}
    |   expression:exp1 MAS:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Arithmetic(info, exp1, exp2, Arithmetic.Operator.SUM);
        :}
    |   expression:exp1 MENOS:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Arithmetic(info, exp1, exp2,
Arithmetic.Operator.SUBTRACTION);
        :}
    |   expression:exp1 MULT:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Arithmetic(info, exp1, exp2,
Arithmetic.Operator.MULTIPLICATION);
        :}
    |   expression:exp1 DIV:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Arithmetic(info, exp1, exp2, Arithmetic.Operator.DIVISION);
        :}
    |   expression:exp1 MODULO:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Arithmetic(info, exp1, exp2, Arithmetic.Operator.MODULE);
        :}
    |   expression:exp1 POTENCIA:op expression:exp2 {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Arithmetic(info, exp1, exp2, Arithmetic.Operator.POWER);
        :}
    |   expression:exp access_list:accessList {:
            NodeInfo infoExp = new NodeInfo(expxleft.getLine(),
expxright.getColumn(), filename);
            RESULT = new StructureAccess(infoExp, exp, accessList);
        :}
    |   MENOS:op expression:exp {:
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new UnarySubtraction(info, exp);
        :} %prec UMENOS
    |   NOT:op expression:exp {:
```

```
            NodeInfo info = new NodeInfo(opxleft.getLine(), opxright.getColumn(),
filename);
            RESULT = new Not(info, exp);
        :} %prec NOT
    |   PAR_IZQ expression:exp PAR_DER {: RESULT = exp; :}
    |   ID:id {:
            NodeInfo info = new NodeInfo(idxleft.getLine(), idxright.getColumn(),
filename);
            RESULT = new Identifier(info, id);
        :}
    |   LIT_ENTERO:lit {:
            NodeInfo info = new NodeInfo(litxleft.getLine(), litxright.getColumn(),
filename);
            RESULT = new Literal(info, TYPE_FACADE.getIntegerType(), lit);
        :}
    |   LIT_DECIMAL:lit {:
            NodeInfo info = new NodeInfo(litxleft.getLine(), litxright.getColumn(),
filename);
            RESULT = new Literal(info, TYPE_FACADE.getNumericType(), lit);
        :}
    |   LIT_BOOLEANO:lit {:
            NodeInfo info = new NodeInfo(litxleft.getLine(), litxright.getColumn(),
filename);
            RESULT = new Literal(info, TYPE_FACADE.getBooleanType(), lit);
        :}
    |   LIT_STRING:lit {:
            NodeInfo info = new NodeInfo(litxleft.getLine(), litxright.getColumn(),
filename);
            RESULT = new Literal(info, TYPE_FACADE.getStringType(), lit);
        :}
    |   NULL:lit {:
            NodeInfo info = new NodeInfo(litxleft.getLine(), litxright.getColumn(),
filename);
            RESULT = new Literal(info, TYPE_FACADE.getStringType(), null);
        :}
;
```

argument_list

Lista de argumentos de una llamada a una funcion

```
argument_list
    ::= argument_list:arguments COMA expression:argument {: RESULT = arguments;
RESULT.add(argument); :}
    |   argument_list:arguments COMA default_exp:argument {: RESULT = arguments;
RESULT.add(argument); :}
    |   expression:argument {: RESULT = new ArrayList<>(); RESULT.add(argument); :}
```

```
    |   default_exp:argument {: RESULT = new ArrayList<>(); RESULT.add(argument);:}
;
```

default_exp

Parametro 'default'.

```
default_exp
    ::= DEFAULT:defaultExp {:
            NodeInfo info = new NodeInfo(defaultExpxleft.getLine(),
defaultExpxright.getColumn(), filename);
            RESULT = new DefaultArgument(info);
        :}
;
```