Manual de la gramática de JavaCC

Opciones

- **JAVA_UNICODE_ESCAPE** : Cuando se establece en verdadero, el analizador generado utiliza un objeto de flujo de entrada que procesa escapes Java Unicode antes de enviar caracteres al administrador de tokens. De manera predeterminada, los escapes Unicode de Java no se procesan. Esta opción se ignora si cualquiera de las opciones USER_TOKEN_MANAGER, USER_CHAR_STREAM está establecida en verdadero.
- **IGNORE_CASE** : Establecer esta opción en verdadero hace que el administrador de tokens generado ignore mayúsculas y minúsculas en las especificaciones de token y los archivos de entrada. Esto es útil para escribir gramáticas para lenguajes como HTML. También es posible localizar el efecto de IGNORE_CASE utilizando un mecanismo alternativo que se describe más adelante.
- **STATIC** : Si es verdadero, todos los métodos y variables de clase se especifican como estáticos en el analizador y el administrador de tokens generados. Esto permite que solo esté presente un objeto analizador, pero mejora el rendimiento del analizador. Para realizar múltiples análisis durante una ejecución de su programa Java, tendrá que llamar al método ReInit () para reinicializar su analizador si es estático. Si el analizador no es estático, puede usar el nuevo operador para construir tantos analizadores como desee. Todos estos pueden usarse simultáneamente desde diferentes hilos.

```
options {
    JAVA_UNICODE_ESCAPE = true;
    IGNORE_CASE = true;
    STATIC = false;
}
```

Codigo de usuario

Codigo utilizado, se creo un nuevo contructor que obtiene un objeto de tipo AritLanguage en cual se guarda una lista de nodos. Ademas se agrego el controlador de tipos.

```
PARSER_BEGIN(Grammar)

public class Grammar {
    private AritLanguage aritLanguage;
    private String filename;

    public Grammar(java.io.Reader reader, @NotNull AritLanguage aritLanguage) {
        this(reader);
        this.aritLanguage = aritLanguage;
        this.filename = aritLanguage.filename;
    }
```

```
    private void addSyntacticError(String message, Token token) {
        this.aritLanguage.addSyntacticError(message, new NodeInfo(token.beginLine,
token.beginColumn, this.filename));
    }

    private static final TypeFacade TYPE_FACADE = TypeFacade.getInstance();
}

PARSER_END(Grammar)
```

Definicion lexica

Espacios en blanco

Caracteres a ignorar durante el analisis

```
/* WHITE SPACE */
SKIP : {
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

Comentarios

Se creo dos estados para cada comentario.

```
/* COMMENTS */
MORE : {
  "#" : IN_SINGLE_LINE_COMMENT
|
  "#*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN : {
    <SINGLE_LINE_COMMENT: (~["\n","\r"])* ("\n"|"\r"|"\r\n")? > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN : {
  <MULTI_LINE_COMMENT: "*#" > : DEFAULT
```

```
}

<IN_SINGLE_LINE_COMMENT, IN_MULTI_LINE_COMMENT>
MORE : {
  < ~[] >
}
```

Palabras reservadas

Se crea los tokens de las palabras reservadas del lenguaje.

```
/* ARIT RESERVED WORDS */
TOKEN : {
  <CONTINUE: "continue">
| <FUNCTION: "function">
| <TDEFAULT: "default">
| <RETURN: "return">
| <SWITCH: "switch">
| <BREAK: "break">
| <WHILE: "while">
| <CASE: "case">
| <ELSE: "else">
| <FOR: "for">
| <IN: "in">
| <DO: "do">
| <IF: "if">
}
```

Literales

Literales del lenguaje arit.

```
/* ARIT LITERALS */
TOKEN : {
  <LIT_ENTERO: (["0"-"9"])+>
|
  <LIT_DECIMAL: (["0"-"9"])+["."](["0"-"9"])+)>
|
  <LIT_BOOLEANO: ("true"|"false")>
|
  <NULL: ("null") >
|
  <LIT_STRING:
    "\""
      (
```

```
        (~["\"","\\","\n","\r"])
      | ("\\" ( ["n","t","r","f","\\","\""] ) )
      )*
    "\""
  >
}
```

Identificadores

```
/* IDENTIFIERS */
TOKEN : {
    <ID: ((["."](["a"-"z","A"-"Z"]|["_"]|["."]))|["a"-"z","A"-"Z"])(["a"-"z","A"-
"Z"]|["0"-"9"]|["_"]|["."])*>
}
```

Separadores y operadores

```
/* SEPARATORS AND OPERATORS */
TOKEN : {
  <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: "}">
}
```

Especificaciones de la gramatica

compilation_unit

Este no terminal devuelve el AST.

```
/**
compilation_unit
    ::= global_statements EOF
    |   EOF
;

global_statements
    ::= global_statements global_statement
    |   global_statement
;
 */
public void compilation_unit() : {
    ArrayList<AstNode> statements = new ArrayList<AstNode>();
    AstNode statement;
} {
    try {
        (
          statement=global_statement()
        {
            if (statement != null) statements.add(statement);
        })* <EOF> {
            aritLanguage.setAstNodes(statements);
        }
    } catch (ParseException e) {
        addSyntacticError(e.getMessage(), e.currentToken);
        Token t;
        do {
            t = getNextToken();
        } while (t.kind != PUNTO_COMA);
        aritLanguage.setAstNodes(statements);
    }
}
```

global_statement

Este no terminal devuelve una sentencia valida en el ambito global. Se utilizo un LOOKAHEAD(7) en las funciones para resolver el conflicto entre asignacion de variables.

```
/**
global_statement
    ::= statement
    |   function_declaration
;
 */
private AstNode global_statement() : {
    AstNode statement;
} {
     (
        LOOKAHEAD(7) statement=function_declaration()
     |  LOOKAHEAD(3) statement=var_declaration()
     |  LOOKAHEAD(2) statement=statement()
     ) { return statement; }
}
```

function_declaration

Este no terminal devuelve la declaracion de una funcion.

```
/**
function_declaration
    ::= id '=' 'function' '(' [ formal_parameter_list ] ')' block
    |   id '=' '(' [ formal_parameter_list ] ')' '=>' block
 */
private Function function_declaration() : {
    ArrayList<FormalParameter> parameters = null;
    Token id;
    Block block;
} {
    id=<ID> <IGUAL> (
            <FUNCTION> <PAR_IZQ> [ parameters=formal_parameter_list() ] <PAR_DER>
        |   <PAR_IZQ> [ parameters=formal_parameter_list() ] <PAR_DER> <LAMBDA>
    ) block=block() [ <PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(id.beginLine, id.beginColumn, this.filename);
        return new Function(info, id.image.toLowerCase(), parameters, block);
    }
}
```

formal_parameter_list

Devuelve una lista de parametros para una funcion.

```
```

```
/**
formal_parameter_list
    ::= formal_parameter_list ',' formal_parameter
    |   formal_parameter
 */
private ArrayList<FormalParameter> formal_parameter_list() : {
    ArrayList<FormalParameter> list = new ArrayList<FormalParameter>();
    FormalParameter parameter;
} {
    parameter=formal_parameter() { list.add(parameter); } (
            <COMA> parameter=formal_parameter() { list.add(parameter); }
        )* { return list; }
}
```

formal_parameter

Devuelve un parametro de una funcion.

```
/**
formal_parameter
    ::= ID
    |   ID '=' expression
;
 */
private FormalParameter formal_parameter(): {
    Token token;
    Expression exp = null;
    NodeInfo info;
} {
    token=<ID> [<IGUAL> exp=expression()] {
        info = new NodeInfo(token.beginLine, token.beginColumn, this.filename);
        return new FormalParameter(info, token.image.toLowerCase(), exp);
    }
}
```

block

Devuelve una lista de sentencias.

```
/*
block
    ::= '{' '}'
    |   '{' block_statements '}'
;
 **/
```

```
private Block block() : {
    Token token;
    ArrayList<AstNode> statements = null;
} {
    token=<LLAVE_IZQ> [ statements=block_statements() ] <LLAVE_DER> {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new Block(info, statements);
    }
}
```

block_statements

Devuelve una lista de sentencias.

```
/*
block_statements
    ::= block_statements statement
    |   statement
;
 */
private ArrayList<AstNode> block_statements() : {
    ArrayList<AstNode> statements = new ArrayList<AstNode>();
    AstNode statement = null;
} {
    try {
    (
        (
            LOOKAHEAD(2) statement=statement()
        |   LOOKAHEAD(4) statement=var_declaration()
        ) {   if (statement != null) statements.add(statement); }
    )+ { return statements; }
    } catch (ParseException e) {
        addSyntacticError(e.getMessage(), e.currentToken);
        Token t;
         do {
            t = getNextToken();
         } while (t.kind != PUNTO_COMA);
         return statements;
    }
}
```

statement

Sentencias validas en cualquier ambito.

```
/**
statement
    ::= break_statement
    |   continue_statement
    |   return_statement
    |   if_statement
    |   switch_statement
    |   while_statement
    |   do_while_statement
    |   for_statement
    |   function_call
;
 */
private AstNode statement() : {
    AstNode statement;
} {
    statement=break_statement()     { return statement; }
|   statement=continue_statement()  { return statement; }
|   statement=return_statement()    { return statement; }
|   statement=if_statement()        { return statement; }
|   statement=switch_statement()    { return statement; }
|   statement=while_statement()     { return statement; }
|   statement=do_while_statement()  { return statement; }
|   statement=for_statement()       { return statement; }
|   statement=function_call()       { return statement; }
}
```

var_declaration

Asignacion de una variable normal o asignacion a una estructura.

```
private AstNode var_declaration() : {
    Token id;
    ArrayList<Access> access_list = new ArrayList<Access>();
    Expression expression;
    Access access;
} {
    id=<ID> (
        access=access() { access_list.add(access); }
    )* <IGUAL> expression=expression() [ <PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(id.beginLine, id.beginColumn, this.filename);
        if (access_list.size() == 0) {
            return new Assignment(info, id.image.toLowerCase(), expression);
        }
        return new StructureAssignment(info, id.image.toLowerCase(), access_list,
expression);
```

```
    }
}
```

access

Acceso a una estructura

```
private Access access() : {
    Token token;
    Expression exp1 = null;
    Expression exp2 = null;
    Access.Type accessType = Access.Type.NORMAL;
} {
    token=<COR_IZQ>
            (
                ( exp1=expression() { accessType = Access.Type.NORMAL; } [
                        <COMA> { accessType = Access.Type.TWO_MATRIX; }
                        [ exp2=expression() { accessType = Access.Type.ONE_MATRIX; }
]
                    ] )
            |   ( <COMA> exp1=expression() { accessType = Access.Type.THREE_MATRIX;
} )
            |   ( <COR_IZQ> exp1=expression() <COR_DER> { accessType =
Access.Type.TWO_LIST; } )
            )
    <COR_DER> {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        if (exp1 != null && exp2 != null) { return new Access(info, exp1, exp2); }
        return new Access(info, exp1, accessType);
    }
}
```

break_statement

Devuelve una sentencia break.

```
/**
break_statement
    ::= BREAK
    |   BREAK ';'
;
 */
private Break break_statement() : {
    Token token;
```

```
} {
    token=<BREAK> [ <PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new Break(info);
    }
}
```

continue_statement

Devuelve una sentencia continue.

```
/**
continue_statement
    ::= CONTINUE
    |   CONTINUE ';'
;
 */
private Continue continue_statement() : {
    Token token;
} {
    token=<CONTINUE> [ <PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new Continue(info);
    }
}
```

return_statement

Devuelve una sentencia return.

```
/**
return_statement
    ::= RETURN
    |   RETURN ';'
    |   RETURN '(' expression ')'
    |   RETURN '(' expression ')' ';'
 */
private Return return_statement() : {
    Token token;
    Expression expression = null;
} {
    token=<RETURN> [ <PAR_IZQ> expression=expression() <PAR_DER> ] [ <PUNTO_COMA> ]
{
```

```
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new Return(info, expression);
    }
}
```

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
    |   if_list 'else' block
;
 */
private IfStatement if_statement() : {
    ArrayList<SubIf> if_list;
    Token token_else;
    Block block_else;
} {
    if_list=if_list() [ token_else=<ELSE> block_else=block() {
        NodeInfo info = new NodeInfo(token_else.beginLine, token_else.beginColumn,
this.filename);
        if_list.add(new SubIf(info, block_else));
    } ] [ <PUNTO_COMA> ] {
        NodeInfo info = if_list.get(0).info;
        return new IfStatement(info, if_list);
    }
}
```

if_list

No terminal que devuelve una lista de if's.

```
/**
if_list
    ::=  'if' '(' expression ')' block
    |   if_list 'else' 'if' '(' expression ')' block
;
 */
private ArrayList<SubIf> if_list() : {
    ArrayList<SubIf> if_list = new ArrayList<SubIf>();
    Token token;
```

```
    Expression expression;
    Block block;
    NodeInfo info;
} {
    token=<IF> <PAR_IZQ> expression=expression() <PAR_DER> block=block() {
        info = new NodeInfo(token.beginLine, token.beginColumn, this.filename);
        if_list.add(new SubIf(info, expression, block));
    }
    (LOOKAHEAD(2) <ELSE> token=<IF> <PAR_IZQ> expression=expression() <PAR_DER>
block=block() {
        info = new NodeInfo(token.beginLine, token.beginColumn, this.filename);
        if_list.add(new SubIf(info, expression, block));
    } )* {
        return if_list;
    }
}
```

switch_statement

No terminal que devuelve una sentencia switch.

```
/**
switch_statement
    ::= 'switch' '(' expression ')' '{' switch_labels '}'
    |   'switch' '(' expression ')' '{' '}'
;
 */
private SwitchStm switch_statement() : {
    Token token;
    Expression expression;
    ArrayList<CaseStm> labels = null;
} {
    token=<SWITCH> <PAR_IZQ> expression=expression() <PAR_DER>
    <LLAVE_IZQ> [ labels=switch_labels() ] <LLAVE_DER> [ <PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new SwitchStm(info, expression, labels);
    }
}
```

switch_labels

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

```
/**
switch_labels
    ::= switch_labels label
    |   label
 */
private ArrayList<CaseStm> switch_labels() : {
    ArrayList<CaseStm> labels = new ArrayList<CaseStm>();
    CaseStm label;
} {
    (label=switch_label() {
        if (label != null) labels.add(label);
    })+ {
        return labels;
    }
}
```

switch_label

No terminal que devuelve una etiqueta.

```
/**
switch_label
    ::= 'case' expression ':' block_statements
    |   'case' expression ':'
    |   'default' ':' block_statements
    |   'default' ':'
;
 */
private CaseStm switch_label() : {
    Token token;
    Expression expression = null;
    ArrayList<AstNode> statements = null;
    NodeInfo info = null;
} {
    token=<CASE> expression=expression() <DOS_PUNTOS> [
statements=block_statements() ] {
        info = new NodeInfo(token.beginLine, token.beginColumn, this.filename);
        return new CaseStm(info, expression, statements);
    }
|   token=<TDEFAULT> <DOS_PUNTOS> [ statements=block_statements() ] {
        info = new NodeInfo(token.beginLine, token.beginColumn, this.filename);
        return new CaseStm(info, null, statements);
    }
}
```

while_statement

No terminal que devuelve una sentencia while.

```
/**
while_statement
    ::= 'while' '(' expression ')' block
;
 */
private WhileStm while_statement() : {
    Token token;
    Expression expression;
    Block block;
} {
    token=<WHILE> <PAR_IZQ> expression=expression() <PAR_DER> block=block() [
<PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new WhileStm(info, expression, block);
    }
}
```

do_while_statement

No terminal que devuelve una sentencia do while.

```
/**
do_while_statement
    ::= 'do' block 'while' '(' expression ')' ';'
    |   'do' block 'while' '(' expression ')'
;
 */
private DoWhileStm do_while_statement() : {
    Token token;
    Block block;
    Expression expression;
} {
    token=<DO> block=block() <WHILE> <PAR_IZQ> expression=expression() <PAR_DER>
[<PUNTO_COMA>] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new DoWhileStm(info, expression, block);
    }
}
```

for_statement

No terminal que devuelve una sentencia for.

```
/*
for_statement
    ::= 'for' '(' ID 'in' expression '') block
;
 **/
private ForStm for_statement() : {
    Token token;
    Token id;
    Expression expression;
    Block block;
} {
    token=<FOR> <PAR_IZQ> id=<ID> <IN> expression=expression() <PAR_DER>
block=block() [ <PUNTO_COMA> ] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new ForStm(info, id.image.toLowerCase(), expression, block);
    }
}
```

function_call

Llamada de una funcion.

```
/**
function_call
    ::= ID '(' ')' ';'
    |   ID '(' ')'
    |   ID '(' argument_list ')' ';'
    |   ID '(' argument_list ')'
;
 */
private FunctionCall function_call() : {
    Token id;
    ArrayList<Expression> argument_list = null;
    Expression argument;
} {
    id=<ID> <PAR_IZQ> [ argument_list=argument_list() ] <PAR_DER> [LOOKAHEAD(2)
<PUNTO_COMA>] {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new FunctionCall(info, id.image.toLowerCase(), argument_list);
    }
}
```

argument_list

Lista de argumentos de una llamada a una funcion

```
/**
argument_list
    ::= argument_list ',' argument
    |   argument
;
 */
private ArrayList<Expression> argument_list() : {
    ArrayList<Expression> argument_list = new ArrayList<Expression>();
    Expression argument;
} {
    argument=argument() { argument_list.add(argument); }
    (<COMA> argument=argument() { argument_list.add(argument); })* {
        return argument_list;
    }
}
```

argument

No terminal argumento de una funcion.

```
/**
argument
    ::= expression
    |   default
;
 */
private Expression argument() : {
    Expression exp;
} {
    exp=expression() { return exp; }
|   exp=default_argument() { return exp; }
}
```

default_argument

Parametro 'default'.

```
/**
default_argument
    ::= default
```

```
;
 */
private DefaultArgument default_argument() : {
    Token token;
} {
    token=<TDEFAULT> {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new DefaultArgument(info);
    }
}
```

expression

Expresiones cada funcion de una expresion sigue la siguiente tabla de precedencia:

Asociatividad	Operador	Nivel
precedence right	'='	1
precedence right	'?', ':'	2
precedence left	'|'	3
precedence left	'&'	4
precedence left	'!=', '=='	5
precedence nonassoc	'>=', '>', '<=', '<'	6
precedence left	'+', '-'	7
precedence left	'*', '/', '%%'	8
precedence left	'^'	9
precedence right	UMENOS ('-'), '!'	10
precedence left	'[', ']', '(', ')'	11

```
/**
expression
    ::= conditional_exp
;
 */
private Expression expression() : {
    Expression exp;
} {
    exp=conditional_exp() { return exp; }
}
```

```
/**
conditional_exp
    ::= conditional_or_exp '?' expression ':' conditional_exp
    |   conditional_or_exp
;
 */
private Expression conditional_exp() : {
    Token token;
    Expression exp;
    Expression exp1;
    Expression exp2;
} {
    exp=conditional_or_exp() [
        token=<INTERROGANTE> exp1=expression() <DOS_PUNTOS> exp2=conditional_exp() {
            NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
            exp = new Ternary(info, exp, exp1, exp2);
        }
    ] {
        return exp;
    }
}


/**
conditional_or_exp
    ::= conditional_and_exp '|' conditional_or_exp
    |   conditional_and_exp
;
 */
private Expression conditional_or_exp() : {
    Token token;
    Expression exp1;
    Expression exp2;
} {
    exp1=conditional_and_exp() (token=<OR> exp2=conditional_and_exp() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        exp1 = new Logical(info, exp1, exp2, Logical.Operator.OR);
    } )* {
        return exp1;
    }
}


private Expression conditional_and_exp() : {
    Token token;
    Expression exp1;
```

```
    Expression exp2;
} {
    exp1=equality_expression() ( token=<AND> exp2=equality_expression() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        exp1 = new Logical(info, exp1, exp2, Logical.Operator.AND);
    } )* {
        return exp1;
    }
}


private Expression equality_expression() : {
    Token token;
    Expression exp1;
    Expression exp2;
    Comparator.Operator operator = Comparator.Operator.UNEQUAL;
} {
    exp1=relational_expression()
    (
        (
            token=<IGUAL_QUE> { operator = Comparator.Operator.EQUAL;}
        |   token=<DIFERENTE_QUE> { operator = Comparator.Operator.UNEQUAL; } )
        exp2=relational_expression() {
            NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
            exp1 = new Comparator(info, exp1, exp2, operator);
        }
    )* {
        return exp1;
    }
}

private Expression relational_expression() : {
    Token token;
    Expression exp1;
    Expression exp2;
    Comparator.Operator operator;
} {
    exp1=additive_expression() (
        LOOKAHEAD(2) (
                        token=<MAYOR_IGUAL_QUE> { operator =
Comparator.Operator.GREATER_THAN_OR_EQUAL_TO; }
                    | token=<MAYOR_QUE>  { operator =
Comparator.Operator.GREATER_THAN; }
                    | token=<MENOR_IGUAL_QUE> { operator =
Comparator.Operator.LESS_THAN_OR_EQUAL_TO; }
                    | token=<MENOR_QUE>  { operator =
Comparator.Operator.LESS_THAN; }
    ) exp2=additive_expression() {
```

```
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        exp1 = new Comparator(info, exp1, exp2, operator);
    }) * {
        return exp1;
    }
}


private Expression additive_expression() : {
    Token token;
    Expression exp1;
    Expression exp2;
    Arithmetic.Operator operator;
} {
    exp1=multiplicative_expression() ( LOOKAHEAD(2)
        (
            token=<MAS>    { operator = Arithmetic.Operator.SUM; }
        |   token=<MENOS>  { operator = Arithmetic.Operator.SUBTRACTION; }
        ) exp2=multiplicative_expression() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        exp1 = new Arithmetic(info, exp1, exp2, operator);
    } )* {
        return exp1;
    }
}


private Expression multiplicative_expression() : {
    Token token;
    Expression exp1;
    Expression exp2;
    Arithmetic.Operator operator;
} {
    exp1=pow_expression() ( LOOKAHEAD(2)
    (
         token=<MULT>   { operator = Arithmetic.Operator.MULTIPLICATION; }
    |    token=<DIV>    { operator = Arithmetic.Operator.DIVISION; }
    |    token=<MODULO> { operator = Arithmetic.Operator.MODULE; }
    ) exp2=pow_expression() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        exp1 = new Arithmetic(info, exp1, exp2, operator);
    } )* {
        return exp1;
    }
}
```

```
private Expression pow_expression() : {
    Token token;
    Expression exp1;
    Expression exp2;
} {
    exp1=unary_expression() ( token=<POTENCIA> exp2=unary_expression() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        exp1 = new Arithmetic(info, exp1, exp2, Arithmetic.Operator.POWER);
    } )* {
        return exp1;
    }
}

private Expression unary_expression() : {
    Expression exp = null;
    Token token;
} {
    token=<MENOS> exp=unary_expression() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new UnarySubtraction(info, exp);
    }
|   exp=unary_expression_not_plus_minus() { return exp; }
}


private Expression unary_expression_not_plus_minus() : {
    Token token;
    Expression exp = null;
} {
    token=<NOT> exp=unary_expression() {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new Not(info, exp);
    }
|   exp=primary_expression() { return exp; }
}

private Expression primary_expression() : {
    ArrayList<Access> accesses = null;
    Access access;
    Expression exp;
} {
    exp=primary_prefix() [ { accesses = new ArrayList<Access>(); } (access=access()
{ accesses.add(access); } )+ ] {
        if (accesses == null) return exp;
        NodeInfo infoExp = exp.info;
        return new StructureAccess(infoExp, exp, accesses);
    }
```

```
}


private Expression primary_prefix() : {
    Expression exp;
} {
    (
        exp=literal()
    |   LOOKAHEAD(<ID> <PAR_IZQ>) exp=function_call()
    |   LOOKAHEAD(<ID>) exp=identifier()
    |   <PAR_IZQ> exp=expression() <PAR_DER>
    ) { return exp; }
}


private Expression identifier() : {
    Token id;
} {
    id=<ID> {
        NodeInfo info = new NodeInfo(id.beginLine, id.beginColumn, this.filename);
        return new Identifier(info, id.image.toLowerCase());
    }
}

private Expression literal() : {
    Token token;
    Object value;
    AritType type = TYPE_FACADE.getUndefinedType();
} {
    (
        token=<LIT_ENTERO> {
            value = 0;
            try {
                type = TYPE_FACADE.getIntegerType();
                value = Integer.parseInt(token.image);
            } catch(NumberFormatException ex) {
                addSyntacticError("Error: Numero " + token.image + " fuera del rango
de un integer.", token);
            }
        }
    |   token=<LIT_DECIMAL> {
            value = 0.0;
            type = TYPE_FACADE.getNumericType();
            try {
                value = Double.parseDouble(token.image);
            } catch (NumberFormatException ex) {
                addSyntacticError("Error: "+ token.image +" fuera del rango de un
decimal.", token);
            }
        }
```

```
    |   token=<LIT_BOOLEANO> {
            value = Boolean.parseBoolean(token.image);
            type = TYPE_FACADE.getBooleanType();
        }
    |   token=<LIT_STRING> {
            String tempImage = token.image.substring(1, token.image.length() - 1);
            tempImage = tempImage.replace("\\\\","\\").replace("\\\"","\"")

.replace("\\n","\n").replace("\\t","\t").replace("\\r","\r");
            value = tempImage;
            type = TYPE_FACADE.getStringType();
        }
    |   token=<NULL> {
            value = null;
            type = TYPE_FACADE.getStringType();
        }
    ) {
        NodeInfo info = new NodeInfo(token.beginLine, token.beginColumn,
this.filename);
        return new Literal(info, type, value);
    }
}
```