Universidad de San Carlos de Guatemala  
Facultad de ingeniería   
Escuela de ciencias y sistemas

Manual de Gramáticas

Fernando Arturo Pensamiento Calderón  
3011-83457-0101  
201602743

# Introducción

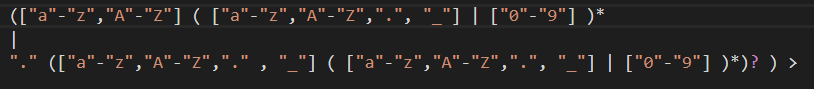
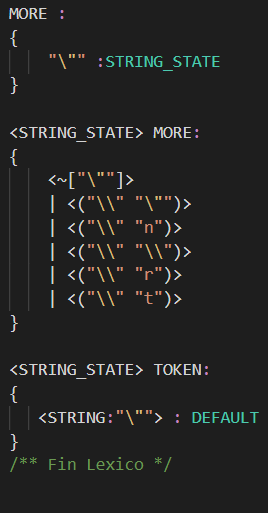
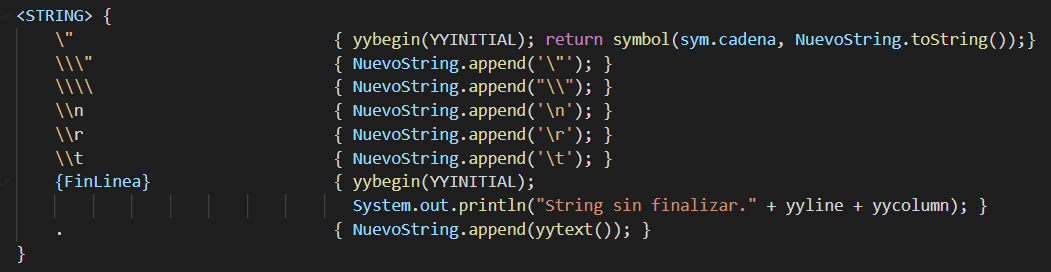
Para realizar este proyecto se necesitó realizar dos gramáticas, una descendente y una ascendente para analizar y ejecutar correctamente el proyecto.

# Objetivos

* Identificar los símbolos terminales.
* Identificar expresiones regulares
* Identificar la precedencia utilizada
* Identificar símbolos no terminales
  + Explicar en que se utilizan cada uno de estos
* Explicar las acciones

# Expresiones regulares utilizadas.

En ambas gramáticas las expresiones regulares utilizadas son las mismas.

* Comentarios
  + Lineal: 
  + Multilinea: 
* Numérico: 
* Decimal: 
* Identificador: 
* Cadena: Esta no es una expresión regular como tal ya que en las dos se utilizaron estados   
   

# Símbolos terminarles del lenguaje.

En ambas gramáticas también se agrega el no terminal <STRING> en javacc y cadena en Flex y cup, estos no se toman en cuenta ya que la ser estados se realizó de forma distinta.

## JavaCC

TOKEN: {

      <NUMERO: (["0"-"9"])+>

    | <DECIMAL: (["0"-"9"])+"."(["0"-"9"])+>

    | <NULL: "null">

    | <TRUE: "true">

    | <FALSE: "false">

    //| <IMPRIMIR: "print">

    | <MIENTRAS: "while">

    | <CASE: "case">

    | <SWITCH: "switch">

    | <FOR: "for">

    | <DO: "do">

    | <SDEFAULT: "default" >

    | <FUNCION: "function">

    | <RETORNO: "Return">

    | <BREAK: "Break">

    | <CONTINUE: "continue">

    | <IN: "in">

    | <SI: "if">

    | <SINO: "else">

    | <PCOMA: ";">

    | <DOSPUNTOS: ":">

    | <PARENI: "(">

    | <PAREND: ")">

    | <CORI: "[">

    | <CORD: "]">

    | <LLAVEI: "{">

    | <LLAVED: "}">

    | <MODULO: "%%">

    | <POTENCIA: "^">

    | <MAS: "+">

    | <PREG: "?">

    | <MENOS: "-">

    | <POR: "\*">

    | <DIV: "/">

    | <IGUAL: "=">

    | <MENORQUE: "<">

    | <MAYORQUE: ">">

    | <MENORIGUAL: "<=">

    | <MAYORIGUAL: ">=">

    | <FLECHA: "=>">

    | <IGUALACION: "==">

    | <DIFERENCIACION: "!=">

    | <COMA: ",">

    | <AND: "&">

    | <OR: "|">

    | <NOT: "!" >

    | <IDENTIFICADOR: (["a"-"z","A"-"Z"] ( ["a"-"z","A"-"Z",".", "\_"] | ["0"-"9"] )\*

                      |

                      "." (["a"-"z","A"-"Z","." , "\_"] ( ["a"-"z","A"-"Z",".", "\_"] | ["0"-"9"] )\*)? ) >

}

## Flex y cup

<YYINITIAL> {

    {entero}            { return symbol(sym.entero, yytext());  }

    {numerico}            { return symbol(sym.numerico, yytext());  }

    "=>"                 { return symbol(sym.flecha, yytext()); }

    "case"                 { return symbol(sym.Case, yytext()); }

    "switch"                 { return symbol(sym.Switch, yytext()); }

    "default"                 { return symbol(sym.Default, yytext()); }

    "null"                 { return symbol(sym.Null, yytext()); }

    "In"                 { return symbol(sym.In, yytext()); }

    "For"                 { return symbol(sym.For, yytext()); }

    "Do"                 { return symbol(sym.Do, yytext()); }

    "Break"                 { return symbol(sym.Break, yytext()); }

    "continue"                 { return symbol(sym.Continue, yytext()); }

    "return"                 { return symbol(sym.Return, yytext()); }

    "function"                 { return symbol(sym.Function, yytext()); }

    "true"                 { return symbol(sym.True, yytext()); }

    "false"                 { return symbol(sym.False, yytext()); }

    "while"                 { return symbol(sym.While, yytext()); }

    "if"                 { return symbol(sym.If, yytext()); }

    "Else"                 { return symbol(sym.Else, yytext()); }

    ","                 { return symbol(sym.coma, yytext()); }

    "="                 { return symbol(sym.igual, yytext()); }

    "=="                 { return symbol(sym.igualigual, yytext()); }

    "!="                 { return symbol(sym.noigual, yytext()); }

    "<="                 { return symbol(sym.menorigual, yytext()); }

    ">="                 { return symbol(sym.mayorigual, yytext()); }

    "<"                 { return symbol(sym.menorque, yytext()); }

    ">"                 { return symbol(sym.mayorque, yytext()); }

    "|"                 { return symbol(sym.or, yytext()); }

    "&"                 { return symbol(sym.and, yytext()); }

    "+"                 { return symbol(sym.mas, yytext()); }

    "-"                 { return symbol(sym.menos, yytext()); }

    "!"                 { return symbol(sym.not, yytext()); }

    "\*"                 { return symbol(sym.por, yytext()); }

    "/"                 { return symbol(sym.division, yytext()); }

    "^"                 { return symbol(sym.potencia, yytext()); }

    "%%"                 { return symbol(sym.modular, yytext()); }

    ";"                 { return symbol(sym.pComa, yytext()); }

    "?"                 { return symbol(sym.preg, yytext()); }

    ":"                 { return symbol(sym.dosPuntos, yytext()); }

    "{"                 { return symbol(sym.llavei, yytext()); }

    "}"                 { return symbol(sym.llaved, yytext()); }

    "["                 { return symbol(sym.cori, yytext()); }

    "]"                 { return symbol(sym.cord, yytext()); }

    "("                 { return symbol(sym.parenI, yytext()); }

    ")"                 { return symbol(sym.parenD, yytext()); }

    \"                  { yybegin(STRING); NuevoString.setLength(0);}

    {iden}            { return symbol(sym.iden, yytext());  }

    {ComentarioLinea}   { /\* ignore \*/}

    {ComentarioMulti}   { /\* ignore \*/}

    {WhiteSpace}        {}

    /\* Cualquier Otro \*/

    .                   {

                          System.out.println("El caracter '"+yytext()+"' no pertenece al lenguaje." + yyline + yycolumn);

                          err.add(  new MiError(yyline, yycolumn, "El caracter '"+yytext()+"' no pertenece al lenguaje.", "lexico")  );

                        }

}

## Cantidad de símbolos terminales:

48, en javacc serían 49 si es que se cuenta el EOF

# Enumeración de no terminales

* Javacc

1. Analizar
2. instrucciones\_cuerpos
3. InstruccionGlobal
4. InstruccionCuerpo
5. lista\_param\_dec\_func
6. Cases
7. Asignacion\_llamadaMetodo
8. ciclos\_if
9. Switch
10. Foreach
11. While
12. do\_while
13. si
14. sino\_si
15. Expresion
16. CondicionOR
17. CondicionAnd
18. ExpresionIgualdad
19. ExpresionRelacional
20. ExpresionAditiva
21. ExpresionMultiplicativa
22. ExpPotencia
23. ExpresionUnaria
24. Primitivo
25. params\_metodo
26. Acceso\_vec\_mat
27. Acceso\_lista

* Flex y cup

1. INICIO
2. INSTRUCCIONES
3. INSTRUCCIÓN
4. CASES
5. SWITCH
6. FOR
7. DO\_WHILE
8. WHILE
9. DECFUNC
10. TRANSFERENCIA
11. LISTA\_PARAMS\_FUN
12. LISTA\_PARAMS\_FUN2
13. PARAMS\_DEC\_FUN
14. FUN\_IF
15. ELSE\_IF
16. ASIGNACION
17. IDEN\_ACCESO
18. IDEN\_ACCESO2
19. LISTACORCHETES
20. LLAMADAFUNC
21. EXPRESION
22. ARITMETICA
23. PRIMITIVO
24. SALIDA

# Explicación de los símbolos no terminales y descripción de sus acciones.

## JavaCC

void Analizar() :

{

  Nodo n;

}

{

  ( n = InstruccionGlobal() { if(n != null){ arr.add(n); }})\* <EOF> { }

}

Este es el método por el cual inicia en javacc, este manda a llamar instrucción global una o más veces y agrega a un arreglo global para que después de analizar se use ese arreglo para poder ejecutar las acciones.

Nodo instrucciones\_cuerpos():

{

  ArrayList<Nodo> arr = new ArrayList<Nodo>(); Nodo n ;

}

{

  ( n =  InstruccionCuerpo() {if( n != null){ arr.add(n); } } )\*

  {

    return new Instrucciones\_cuerpo(0 , 0 , arr);

  }

}

Instrucciones cuerpo es bastante similar al de analizar con la única diferencia que este utiliza un arreglo local para después ser agregado un nodo y recorrido la cantidad de veces que sea necesaria.

Nodo  InstruccionGlobal() :

  {

    Nodo n;

  }

  {

    (

      n = InstruccionCuerpo()

    ){ return n; }

  }

En este método solo se manda a llamar a instrucciones cuerpo, esto fue así porque al principio pensaba hacer que solo en el global se pudieran crear funciones pero ya no se realizó.

  Nodo InstruccionCuerpo():

  {

    Nodo n= null;

  }

  {

    try{

      (

        n = Cases()

        |

        <CONTINUE> { n = new Continuar( token.beginLine, token.beginColumn );} (<PCOMA>)?

        |

        <BREAK>    { n = new Parar( token.beginLine, token.beginColumn ); }  (<PCOMA>)?

        |

        n = ciclos\_if()

        |

        <RETORNO> (<PARENI> n =  Expresion()  <PAREND>)? (<PCOMA>)? {n = new Retorno(token.beginLine, token.beginColumn , n);}

        |

        n = Asignacion\_llamadaMetodo()

      )  {return n;   }

    }catch(ParseException x){

        //System.out.println(x.toString());

        Token terr = x.currentToken.next;

        System.out.println("Token \"" + terr.image + "\" no esperado en f. " + terr.beginLine + "  c." + terr.beginColumn );

        miErr.add(  new MiError(terr.beginLine, terr.beginColumn, "No se esperaba  '"+ terr.image +"'", "sintactico")  );

        Token t;

        do{

          t = getNextToken();

        }while(!(t.kind == PCOMA || t.kind == LLAVED || t.kind == EOF));

    }catch(Exception e){

      System.out.println(e.toString());

    }

    {return n;}

  }

Instrucciones cuerpo es el método que se encarga de direccionar a cualquier cosa que se pueda realizar en el lenguaje y tiene la recuperación de errores. Este solo llama diferentes métodos y retorna lo que estos retornen

  ArrayList<Nodo> lista\_param\_dec\_func() :

  {

    ArrayList<Nodo> arr = new ArrayList<Nodo>();

    Nodo n = null , n1; Token t;

  }

  {

    ( n1 = Expresion()

                (t = <IGUAL> n =  Expresion(){ n1 = new e\_e(t.beginLine , t.beginColumn , n1 , n);} )? { arr.add(n1); }

    (<COMA> n1 = Expresion()

            (t = <IGUAL> n = Expresion() {n1 = new e\_e(t.beginLine , t.beginColumn , n1 , n);; } )? { arr.add(n1);  }  )\* )?

    { return arr; }

  }

En este método es el que se encarga de listar los parámetros que se utilizan en la declaración de funciones, aunque al final para que no fuera ambigua es el que se usa adentro de paréntesis, esto hace que se permitan bastantes cosas incorrectas sintácticamente que toco arreglar en la ejecución

Nodo Cases():

  {

    Nodo n ,n1 , n2; Token t;

  }

  {

    (

    t = <CASE> n1= Expresion() <DOSPUNTOS>  {n = new Case(t.beginLine , t.beginColumn , n1); }

    |

    t = <SDEFAULT> <DOSPUNTOS>    {n = new Default(t.beginLine , t.beginColumn); }

    ){

      return n;

    }

  }

En este método solo se verifican los diferentes casos que pueden venir en el switch. Y de igual forma crean un nodo de su respectivo elemento y lo retornan.

Nodo Asignacion\_llamadaMetodo() :

  {

    Nodo n1 , n2 , n3 , n; Token t, t2; ArrayList<Nodo> arr;

  }

  {

    (

      t2 = <IDENTIFICADOR> { n1 = new Iden(t2.beginLine, t2.beginColumn , t2.image);  }

      (

        //Asignacion    a = 2;

        t=<IGUAL>  (

            n2 = Expresion()  (

                      <PCOMA>

                      |

                      t = <FLECHA> <LLAVEI>

                      n3 =   instrucciones\_cuerpos()

                      <LLAVED>   {

                              arr = n2.hijos;

                              if(!(n2 instanceof Paren)){

                                miErr.add(new MiError(t.beginLine , t.beginColumn, "no se esparaba este componente => ", "lexico"));

                                  System.out.println("no se esparaba este componente => ");

                                  return new Asignacion\_funcion(t.beginLine , t.beginColumn , arr , t2.image , true);

                              }

                              arr.add(n3);

                              return new Asignacion\_funcion(t.beginLine , t.beginColumn , arr , t2.image);

                      }   //=>

                    )?  //Fin asginacion

            {    return new Asignacion(t.beginLine, t.beginColumn , n1 , n2); }

            |

              <FUNCION> <PARENI> arr = lista\_param\_dec\_func()  <PAREND> <LLAVEI>

                n2 =   instrucciones\_cuerpos()

              <LLAVED>

              { arr.add(n2);  return new Asignacion\_funcion(t2.beginLine , t2.beginColumn , arr , t2.image); }

            )

         |

         //asignacion de a[1] = 2;

         <CORI> n1 = Acceso\_vec\_mat(n1)

         t=<IGUAL>  n2 = Expresion()  (<PCOMA>)?  //Fin asginacion

         { return new Asignacion(t.beginLine, t.beginColumn , n1 , n2); }

         //lamada a metodo

         |

         arr = params\_metodo() (<PCOMA>)?

         { return new Llamada\_metodo(t2.beginLine , t2.beginColumn , arr , t2.image ); }

         //fin llamada a metodo

      )

    )

  }

Este método se encarga de las asignaciones, tanto de los accesos a matriz del lado de la asignación como la asignación de los métodos y esas cosas. Cabe mencionar que es uno de los pocos métodos que necesite utilizar atributos heredados. Para los accesos a objetos.

  Nodo ciclos\_if():

  {

    Nodo n ;

  }

  {

    (

     n =  si()

     |

     n = While()

     |

     n = do\_while()

     |

     n = foreach()

     |

     n = Switch()

    )

    {

      return n;

    }

  }

En este método se manda a llamar a la declaración de diferentes métodos.

  Nodo Switch():

  {

    Nodo n , n1, n2; Token t;

  }

  {

     t = <SWITCH> <PARENI> n1 =   Expresion() <PAREND> <LLAVEI> n2 =  instrucciones\_cuerpos() <LLAVED>

     {

        return new Switch(t.beginLine , t.beginColumn , n1 , n2);

     }

  }

Reconoce las sentencias del switch y devuelve un nodo de tipo switch.

 Nodo foreach():

  {

    Nodo n , n1 , n2; Token t , t1;

  }

  {

    t1 = <FOR> <PARENI> t = <IDENTIFICADOR> <IN> n1 =  Expresion() <PAREND> <LLAVEI>

      n2 =   instrucciones\_cuerpos()

    <LLAVED>

    {

      return new For(t1.beginLine , t1.beginColumn , n1 , n2 , t.image);

    }

  }

Este método reconoce el for y devuelve un nodo de tipo for.

  Nodo While():

  {

    Nodo n, n1; Token t;

  }

  {

    t = <MIENTRAS> <PARENI> n= Expresion()  <PAREND>

    <LLAVEI> n1= instrucciones\_cuerpos()  <LLAVED>

    {

      return new While(t.beginLine , t.beginColumn , n , n1);

    }

  }

El método While reconoce la sentencia while y devuelve un nodo de tipo While

  Nodo do\_while():

  {

    Nodo n, n1; Token t;

  }

  {

    t = <DO> <LLAVEI> n =  instrucciones\_cuerpos()  <LLAVED> <MIENTRAS> <PARENI> n1 = Expresion() <PAREND> (<PCOMA>)?

    {

      return new Do\_while(t.beginLine , t.beginColumn , n , n1);

    }

  }

El método do\_while reconoce el do while y devuelve un nodo de tipo do while

  Nodo si():

  {

    Nodo n; Token t;

    ArrayList<Nodo> arr = new ArrayList<Nodo>();

  }

  {

    (

      t = <SI> <PARENI> n = Expresion() { arr.add(n); }  <PAREND> <LLAVEI> n =  instrucciones\_cuerpos() {arr.add(n); }

      <LLAVED>  ( n =  sino\_si()  { arr.add(n); }  )?

    ){

      return new If(t.beginLine , t.beginColumn ,arr );

    }

  }

EL método si reconoce la instrucción si, revisa si hay más si\_no\_si o si\_no anidados y devuelve un nodo de tipo si

Nodo sino\_si():

  {

    Token t;

    Nodo n , n2 ;

  }

  {

    t = <SINO> (

      n = si()

      |

      <LLAVEI> n =  instrucciones\_cuerpos()   <LLAVED>

    ){

      return n;

    }

  }

EL método si\_no\_si reconoce los si\_no\_si o si\_no anidados que el si contenga.

Nodo Expresion() :

  {

    Nodo n , n2 , n3; Token t;

    ArrayList<Nodo> arr = new ArrayList<Nodo>();

  }

  {

      n = CondicionOR() ( t = <PREG> n2 = Expresion() <DOSPUNTOS> n3 = Expresion()

            {arr.add(n); arr.add(n2); arr.add(n3); n = new Ternario(t.beginLine , t.beginColumn , arr); }

          )?

      {

        return n;

      }

  }

El nodo expresión es que se encarga de realizar todo lo relacionado con expresión y en este es donde se realiza la operación ternaria.

  Nodo CondicionOR() :

  {

    Nodo n , n2;

  }

  {

    n = CondicionAnd()

    (

        <OR> n2 = CondicionAnd() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.or); }

    )\*

    {

      return n;

    }

  }

Se encarga de recibir todos los OR que puedan existir y manda a llamar a la condición AND, crear los nodos de tipo Or.

  Nodo CondicionAnd() :

  {

    Nodo n , n2;

  }

  {

      n = ExpresionIgualdad()

      (

          <AND> n2 = ExpresionIgualdad() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.and); }

      )\*

      {

        return n;

      }

  }

Recibe todas las and anidadas, crea los nodos de and y llama a expresión igualdad

Nodo ExpresionRelacional() :

  {

    Nodo n , n2;

  }

  {

     n = ExpresionAditiva()

      (

          <MAYORQUE> n2 = ExpresionAditiva() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.mayorque);}

          |

          <MENORQUE> n2 = ExpresionAditiva() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.menorque);}

          |

          <MAYORIGUAL> n2 = ExpresionAditiva() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.mayorigual);}

          |

          <MENORIGUAL> n2 = ExpresionAditiva() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.menorigual);}

      )\*

      {

        return n;

      }

  }

En expresión igualdad se crean los nodos de igualdad y desigualdad y manda a llamar a expresión aditiva.

 Nodo  ExpresionAditiva() :

  {

    Nodo n , n2;

  }

  {

      n = ExpresionMultiplicativa()

      (

          <MAS> n2=  ExpresionMultiplicativa() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.mas); }

          |

          <MENOS> n2 = ExpresionMultiplicativa() {n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.menos); }

      )\*

      {

        return n;

      }

  }

En la expresión aditiva se realizan los nodos de mas y menos y se llaman a la expresión multiplicativa

 Nodo ExpresionMultiplicativa() :

  {

    Nodo n , n2;

  }

  {

      n = ExpPotencia()

      (

          <POR> n2 = ExpPotencia() {n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.por); }

          |

          <DIV> n2 = ExpPotencia() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.div);}

      )\*

      {

        return n;

      }

  }

Realiza los nodos de por y dividir y manda a llamar a expotencia

  Nodo ExpPotencia() :

  {

    Nodo n , n2;

  }

  {

      n = ExpresionUnaria()

      (

          <POTENCIA> n2 = ExpresionUnaria() {n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.potencia); }

          |

          <MODULO> n2 = ExpresionUnaria() { n = new OperadorBinario(token.beginLine , token.beginColumn, n , n2, Operando.modulo);}

      )\*

      {

        return n;

      }

  }

Crea los nodos de potencia y modulo y manda a llamar a las expresiones unarias.

Nodo ExpresionUnaria() :

  {

    Nodo n ;

  }

  {

    (

      <MENOS> n = ExpresionUnaria() {n = new OperadorUnario(token.beginLine , token.beginColumn, n, Op.neg);  }

      |

      <NOT> n = ExpresionUnaria() { n = new OperadorUnario(token.beginLine , token.beginColumn, n, Op.not);  }

      |

      n= Primitivo() {}

    ){

      return n;

    }

  }

Este realiza los nodos de operadorUnario y manda a llamar a los primitivos.

  Nodo Primitivo() :

  {

    Nodo n; Token t; ArrayList<Nodo> arr = new ArrayList<Nodo>();

  }

  {

    (

      <NUMERO> { n = new Primitivo( token.beginLine, token.beginColumn , Tipos.entero , Double.parseDouble(token.image)  ); }

      |

      <DECIMAL> {n = new Primitivo( token.beginLine, token.beginColumn , Tipos.numerico , Double.parseDouble(token.image)  );  }

      |

      t = <STRING> {n = new Primitivo( token.beginLine, token.beginColumn , Tipos.cadena , t.image.substring(1,t.image.length()-1).replace("\\\"", "\"").replace("\\\\" , "\\").replace("\\n" , "\n").replace("\\r", "\r").replace("\\t" , "\t")  ); }

      |

      <TRUE>   { n = new Primitivo( token.beginLine, token.beginColumn , Tipos.booleano , true  ); }

      |

      <FALSE>  {n = new Primitivo( token.beginLine, token.beginColumn , Tipos.booleano , false  ); }

      |

      t = <IDENTIFICADOR> { n = new Iden(t.beginLine, t.beginColumn , t.image); }

          (

                <CORI> n = Acceso\_vec\_mat(n)

                |

                arr = params\_metodo()  { n = new Llamada\_metodo(t.beginLine, t.beginColumn , arr, t.image);  }

          )?

      |

      <NULL> {n  = new Primitivo(token.beginLine , token.beginColumn , Tipos.nulo, "");}

      |

      t = <PARENI> arr= lista\_param\_dec\_func() <PAREND>   { n = new Paren(t.beginLine , t.beginColumn , arr);  }

    )

    {

      return n;

    }

  }

En primitivo se realizan los primitivos, las llamadas a funciones, los accesos y otras cosas.

  ArrayList<Nodo> params\_metodo():

  {

    ArrayList<Nodo> arr = new ArrayList<Nodo>(); Nodo n; Token t;

  }

  {

    <PARENI> ( (n = Expresion() {arr.add(n);} | t =  <SDEFAULT> { n = new Default(t.beginLine , t.beginColumn); arr.add(n); }  ) (

      <COMA> ( n =  Expresion() {arr.add(n);} | t =  <SDEFAULT> { n = new Default(t.beginLine , t.beginColumn); arr.add(n); } ) )\*  )?  <PAREND>

    { return arr;  }

  }

Este es el método que sirve para las llamadas a funciones. El cual devuelve una lista de expresiones.

Nodo Acceso\_vec\_mat(Nodo n):   //atributo heredado

  {

    Nodo  n1 , n2; Token t;

    ArrayList<Nodo> arr = new ArrayList<Nodo>();

  }

  {

    (

      n1 = Expresion() ( <CORD> { arr.add(n); arr.add(new Acceso(n1.fila , n1.columna , n1 , false ) ); }  ( n1 =  Acceso\_lista()  { arr.add(n1); }  )\*

      {return new Var\_acceso(n.fila, n.columna , arr);}  /\* [E]([E] | [[E]])\*  \*/

                    | t =  <COMA> (

                            n2 = Expresion() <CORD> {n1 =  new AccesoMatriz(t.beginLine, t.beginColumn , n1 , n2 );

                                                    return new Var\_acceso(n.fila, n.columna , n , n1); }   /\* [E,E] \*/

                            |

                            <CORD>    {n1 =  new AccesoMatriz(t.beginLine, t.beginColumn , n1 , true );

                                        return new Var\_acceso(n.fila, n.columna , n , n1);}           /\* [E,] \*/

                            )

                  )

      |

      t = <COMA> (

       n1 =  Expresion() <CORD>   {n1 =  new AccesoMatriz(t.beginLine, t.beginColumn , n1 , false );

                                    return new Var\_acceso(n.fila, n.columna , n , n1);} /\* [,E] \*/

      )

    |

    <CORI> n1 = Expresion() { arr.add(n);  arr.add(new Acceso(n1.fila , n1.columna , n1 , true ) ); } <CORD> <CORD> (n1 = Acceso\_lista() {arr.add(n1);  } )\*

    { return new Var\_acceso(n.fila, n.columna , arr);}    /\* [[E]]([E] | [[E]])\*  \*/

    )

    {

      return n;

    }

  }

Es el que se encarga de acceso a matrices y si llama a arreglos manda a llamar acceso a lista.

  Nodo Acceso\_lista():

  {

    Nodo n1;

  }

  {

    (

      <CORI>

        ( n1 =  Expresion() <CORD> { n1 =  new Acceso(n1.fila , n1.columna , n1 , false );  }

            |

        <CORI> n1 =  Expresion() <CORD> <CORD> { n1 = new Acceso(n1.fila , n1.columna , n1 , true ); }

      )

    )

    {

      return n1;

    }

  }

Se encarga de devolver nodos de acceso.

### Precedencia

1. Ternario
2. Or
3. And
4. == y !=
5. + y -
6. \* y /
7. ^ y %%
8. – (unario) y not
9. Primitivos

## Flex y cup

INICIO ::= INSTRUCCIONES:e {: miarr = e;  :}

         ;

Es donde inicia todo, manda a llamar instrucciones y luego lo agrega a un arraylist que se accede después de analizar todo.

INSTRUCCIONES ::= INSTRUCCIONES:arr INSTRUCCION:a    {: if(a == null){ RESULT = arr;  } else{  arr.add(a); RESULT = arr;} :}

                |                     {: RESULT = new ArrayList<Nodo>(); :}

                ;

Este no terminal se encarga de hacer una lista de instrucción. Y devuelve un arraylist de nodos.

INSTRUCCION ::= LLAMADAFUNC:a        SALIDA{: RESULT = a; :}

                | ASIGNACION:a       SALIDA{: RESULT = a; :}

                | FUN\_IF:a                 {: RESULT = a; :}

                | DECFUNC:a                {: RESULT = a; :}

                | WHILE:a                  {: RESULT = a; :}

                | FOR:a                    {: RESULT = a; :}

                | DO\_WHILE:a         SALIDA{: RESULT = a; :}

                | TRANSFERENCIA:a    SALIDA{: RESULT = a; :}

                | SWITCH:a                 {: RESULT = a; :}

                | CASES:a                  {: RESULT = a; :}

                | error:e            pComa {: RESULT = null; :}

                | error:e            llaved{: RESULT = null; :}

              ;

En este no terminal se manda a llamar a todo lo que se puede realizar en la gramática y es el encargado de arreglar errores.

CASES ::= Case:a EXPRESION:n dosPuntos              {: RESULT = new Case(aleft , aright , n); :}

        | Default:a dosPuntos                       {: RESULT = new Default(aleft , aright);  :}

        ;

Manda a llamar a los nodos que se utilizan en el switch.

SWITCH ::= Switch:a parenI EXPRESION:n1 parenD llavei INSTRUCCIONES:arr  llaved {: Nodo n2 = new Instrucciones\_cuerpo(0,0,arr);

                                                                RESULT = new Switch(aleft , aright ,n1 , n2); :} ;

Reconoce el switch y crea el nodo de switch

FOR ::= For parenI iden:a In EXPRESION:n1 parenD llavei  INSTRUCCIONES:arr llaved {: Nodo n2 = new Instrucciones\_cuerpo(0,0,arr);

                                                        RESULT = new For(aleft , aright , n1 , n2 , a);        :}

        ;

Reconoce el for y crea el nodo de for

DO\_WHILE ::= Do:a llavei  INSTRUCCIONES:arr llaved While parenI EXPRESION:n1 parenD {: Nodo n = new Instrucciones\_cuerpo(0,0,arr);

                                                                        RESULT = new Do\_while(aleft , aright , n , n1);:}

        ;

Reconoce el do while y crea el nodo do\_while

WHILE ::= While:a parenI EXPRESION:n parenD llavei INSTRUCCIONES:arr llaved {: Nodo n1 = new Instrucciones\_cuerpo(0,0,arr);

                                                                        RESULT = new While(aleft , aright , n , n1); :}

        ;

Reconoce el while y crea el nodo while

DECFUNC ::= iden:a igual Function parenI LISTA\_PARAMS\_FUN:arr  parenD llavei INSTRUCCIONES:arr2  llaved

                        {:

                        Nodo n = new Instrucciones\_cuerpo(0,0 , arr2);

                        arr.add(n);

                        RESULT = new Asignacion\_funcion(aleft , aright , arr , a); :}

                | iden:a igual parenI LISTA\_PARAMS\_FUN:arr  parenD flecha  llavei INSTRUCCIONES:arr2  llaved

                {:

                        Nodo n = new Instrucciones\_cuerpo(0,0 , arr2);

                        arr.add(n);

                        RESULT = new Asignacion\_funcion(aleft , aright , arr , a);

                :}

                | iden:a igual parenI iden:n parenD flecha llavei INSTRUCCIONES:arr2 llaved

                {:

                        Nodo n1 = new Instrucciones\_cuerpo(0,0,arr2);

                        ArrayList<Nodo> arr = new ArrayList<Nodo>();

                        arr.add(new Iden(nleft , nright , n)); arr.add(n1);

                        RESULT = new Asignacion\_funcion(aleft , aright , arr , a);

                :}

                ;

En declaración función se realizan las funciones.

TRANSFERENCIA ::= Return:a                         {: RESULT = new Retorno(aleft, aright , null); :}

        | Return:a parenI EXPRESION:n parenD {: RESULT = new Retorno(aleft, aright , n);    :}

        | Break:a        {:RESULT = new Parar( aleft, aright ); :}

        | Continue:a    {:RESULT = new Continuar( aleft, aright ); :}

        ;

Manda a llamar los nodos de transferencia, los cuales son return, break y continue

LISTA\_PARAMS\_FUN  ::=  LISTA\_PARAMS\_FUN2:arr {:RESULT = arr; :}

                        | {: RESULT = new ArrayList<Nodo>(); :}

                        ;

En este se manda a llamar a la lista de parámetros o vacío para los parámetros.

LISTA\_PARAMS\_FUN2 ::=  LISTA\_PARAMS\_FUN2:arr coma PARAMS\_DEC\_FUN:n {: arr.add(n); RESULT = arr; :}

                        | PARAMS\_DEC\_FUN:n {: ArrayList<Nodo> arr = new ArrayList<Nodo>(); arr.add(n); RESULT = arr; :}

                        ;

En este método se crean las listas de parámetros y se devuelve un arraylist.

PARAMS\_DEC\_FUN    ::= EXPRESION:n                   {: RESULT =  n; :}

                |     EXPRESION:n igual:a EXPRESION:n1 {: RESULT =  new e\_e(aleft , aright , n, n1); :}

                |     Default:a     {:RESULT = new Default(aleft, aright); :}

                ;

Se crean los parámetros de funciones, en este caso acepta expresión = expresión y lo verifique en ejecución.

FUN\_IF ::= If:a parenI EXPRESION:n parenD llavei  INSTRUCCIONES:arr  llaved ELSE\_IF:n2 {:  Nodo n1 = new Instrucciones\_cuerpo(0 , 0 , arr);

                                                                                        /\* if operador expresiones\_cuerpo\*/

                                                                                        ArrayList<Nodo> arr2 = new ArrayList<Nodo>();

                                                                                        arr2.add(n); arr2.add(n1);

                                                                                        if(n2 != null){ arr2.add(n2); }

                                                                                        RESULT = new If(aleft , aright ,arr2 );

                                                                                        :}

Reconoce el if y devuelve su nodo y manda a llamar al los si\_no y si\_no\_si anidados

ELSE\_IF ::= Else llavei INSTRUCCIONES:arr llaved{: RESULT = new Instrucciones\_cuerpo(0 , 0 , arr);  :}

        | Else FUN\_IF:a  {: RESULT = a; :}

         | {: RESULT = null;:} ;

Se crean los if anidados y el else

ASIGNACION ::=

        iden:i igual:a EXPRESION:n2        {: Nodo n  =  new Iden(ileft , iright , i);

                                              RESULT  =  new Asignacion(aleft, aright , n , n2); :}

        |

        IDEN\_ACCESO:n igual:a EXPRESION:n2 {: RESULT  =  new Asignacion(aleft, aright , n , n2); :}

        ;

Se encarga de las asignaciones, tanto en accesos como normales.

IDEN\_ACCESO ::= IDEN\_ACCESO2:n        {: RESULT = n;  :}

                | LISTACORCHETES:arr  {: RESULT = new Var\_acceso(arr.get(0).fila, arr.get(0).columna , arr);  :}

        ;

Manda a llamar a los accesos de matrices(iden\_acceso) y a los de arreglos o vectores o listas.

IDEN\_ACCESO2 ::= iden:a  cori:t EXPRESION:n1 coma EXPRESION:n2 cord {: Nodo n = new AccesoMatriz(tleft , tright , n1 , n2 );

                                                                       RESULT = new Var\_acceso(tleft , tright , new Iden(aleft , aright , a) , n); :}

                |iden:a  cori:t EXPRESION:n1  coma cord        {: Nodo n = new AccesoMatriz(tleft , tright , n1 , true );

                                                                       RESULT = new Var\_acceso(tleft , tright , new Iden(aleft , aright , a) , n); :}

                |iden:a  cori:t coma EXPRESION:n1  cord        {: Nodo n = new AccesoMatriz(tleft , tright , n1 , false );

                                                                       RESULT = new Var\_acceso(tleft , tright , new Iden(aleft , aright , a) , n); :}

                ;

Se encarga de los accesos a matrices.

LISTACORCHETES ::= LISTACORCHETES:arr cori:a EXPRESION:n cord {: arr.add(new Acceso(aleft , aright , n , false )); RESULT = arr; :}

                | LISTACORCHETES:arr cori:a cori EXPRESION:n cord cord {: arr.add(new Acceso(aleft , aright , n , true )); RESULT = arr; :}

                | iden:a  cori EXPRESION:n2 cord  {:ArrayList<Nodo> arr = new ArrayList<Nodo>();

                        Nodo n =  new Iden(aleft , aright , a);

                        arr.add(n);arr.add(new Acceso(aleft , aright , n2 , false ));   RESULT  = arr;  :}

                | iden:a  cori cori EXPRESION:n2 cord cord  {:ArrayList<Nodo> arr = new ArrayList<Nodo>();

                        Nodo n =  new Iden(aleft , aright , a);

                        arr.add(n);arr.add(new Acceso(aleft , aright , n2 , true ));   RESULT  = arr;  :}

                        ;

Se encarga de los accesos a arreglos.

LLAMADAFUNC ::= iden:a parenI LISTA\_PARAMS\_FUN:arr parenD  {: RESULT = new Llamada\_metodo(aleft, aright , arr, a);    :}

           ;

Se encarga de las llamadas de funciones.

EXPRESION ::=  ARITMETICA:e                 {: RESULT = e; :}

           |   PRIMITIVO:e                  {: RESULT = e; :}

           |   LLAMADAFUNC:e                {: RESULT = e; :}

           |   EXPRESION:n preg:a EXPRESION:n1 dosPuntos EXPRESION:n2

                {: ArrayList<Nodo> arr = new ArrayList<Nodo>();

                arr.add(n); arr.add(n1); arr.add(n2);

                RESULT = new Ternario(aleft , aright , arr);

                :}

           ;

Son las operaciones que se pueden realizar. O llamadas a funciones y ternarios.

ARITMETICA ::= menos:a  EXPRESION:n                    {:  RESULT = new OperadorUnario(aleft  , aright, n, Op.neg);                     :} %prec menos

                | not:a  EXPRESION:n                   {:  RESULT = new OperadorUnario(aleft  , aright, n, Op.not);                     :}

                | EXPRESION:n mas:a EXPRESION:n2       {:  RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.mas);          :}

                | EXPRESION:n menos:a EXPRESION:n2     {:  RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.menos);        :}

                | EXPRESION:n por:a EXPRESION:n2       {:  RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.por);          :}

                | EXPRESION:n division:a EXPRESION:n2  {:  RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.div);          :}

                | EXPRESION:n potencia:a EXPRESION:n2  {:  RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.potencia);     :}

                | EXPRESION:n modular:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.modulo);         :}

                | EXPRESION:n igualigual:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.comparacion);         :}

                | EXPRESION:n noigual:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.desigualdad);         :}

                | EXPRESION:n mayorque:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.mayorque);         :}

                | EXPRESION:n mayorigual:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.mayorigual);         :}

                | EXPRESION:n menorque:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.menorque);         :}

                | EXPRESION:n menorigual:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.menorigual);         :}

                | EXPRESION:n or:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.or);         :}

                | EXPRESION:n and:a EXPRESION:n2  {: RESULT = new OperadorBinario(aleft , aright, n , n2, Operando.and);         :}

             ;

Analiza y crea nodos de las funciones aritméticas.

PRIMITIVO ::= numerico:e  {: RESULT = new  Primitivo(eleft , eright , Tipos.numerico,  Double.parseDouble(e) ); :}

            | cadena:e  {:   RESULT = new  Primitivo(eleft , eright , Tipos.cadena,  e ); :}

            | entero:e  {:   RESULT = new  Primitivo(eleft , eright , Tipos.entero,  e ); :}

            | True:e  {:   RESULT = new  Primitivo(eleft , eright , Tipos.booleano,  true ); :}

            | False:e  {:   RESULT = new  Primitivo(eleft , eright , Tipos.booleano,  false ); :}

            | IDEN\_ACCESO:n   {:  RESULT = n; :}

            |iden:a               {: RESULT =  new Iden(aleft , aright , a); :}

            | Null:e {: RESULT = new  Primitivo(eleft , eright , Tipos.nulo,  "" ); :}

             |   parenI EXPRESION:e parenD    {: RESULT = e; :}

           ;

Datos primitivos, accesos a vectores o matrices y llamadas a funciones.

SALIDA ::= pComa | ;

Ya que es opcional el punto y coma se realizo una producción para hacer eso.

### Precedencia

precedence right igual;

precedence right preg , dosPuntos;

precedence left or;

precedence left and;

precedence left igualigual, noigual;

precedence left mayorque, mayorigual, menorque , menorigual;

precedence left mas, menos;

precedence left por, division, modular;

precedence left potencia;

precedence left parenI, parenD;

precedence right not;