

Gramáticas

JFlex y Cup


El archivo de jflex cuenta con un área de declaración de políticas en ellas vamos a declarar todas las importaciones necesarias y directivas que se utilizaran para los tokens.

```
package AnalizadorA;
import Reportes.Errores;
import java_cup.runtime.*;
import java.util.ArrayList;
import Reportes.Errores.TipoError;
import java.util.LinkedList;
%%
%cup
%class scanner
%public
%line
%char
%column
%full
%ignorecase
%{
    public LinkedList<Errores>listaerrores=new LinkedList<>();;
}%
/*
```

Se declara que se necesita la línea, columna y que sea ignore case. Declaramos una lista que contiene errores para almacenar errores léxicos. Luego se viene el área de declaración de tokens léxicos.

```
/*
PAR_A="("
PAR_C=")"
COR_A="["
COR_C="]"
LLAV_A="{ "
LLAV_C="}"
IGUAL="="
PYCOMA=",";
MAS="+"
MENOS="-"
POR="*"
DIV="/"
POTENCIA="^"
MODULO="%"
IGUAL_IGUAL=="=="
DISTINTO!="!="
MAYOR_I=">="
MENOR_I="<="
MAYOR=">"
MENOR="<"
AND="&"
OR="|"
NOT="!"
PREGUNTA="?"
DOSPUNTOS=":"
COMA=","
/*
```

Área de palabras reservadas.

```
*/  
IN="in"  
FOR="for"  
TRUE="true"  
FALSE="false"  
IF="if"  
ELSE="else"  
SWITCH="switch"  
CASE="case"  
BREAK="break"  
DEFAULT="default"  
FUNCTION="function"  
RETURN="return"  
WHILE="while"  
DO="do"  
CONTINUE="continue"  
/*  
  
*/  
NUMERIC=[0-9]+ "." [0-9]+  
INTEGER=[0-9]+  
NULO="null"  
STRING=\"([^\"])*\"  
/*
```

Luego se declara la parte de acciones en donde se retornan los valores de los token también imprimimos el token que esta siendo analizado para identificar errores en el análisis lexico.

```
%%
<YYINITIAL>{PAR_A}      {System.out.println("Token "+yytext()+" reconocido"); return new Symbol(sym.PAR_A,yyline,yycolumn,yytext());}
<YYINITIAL>{PAR_C}      {System.out.println("Token "+yytext()+" reconocido"); return new Symbol(sym.PAR_C,yyline,yycolumn,yytext());}
<YYINITIAL>{PAR_A}      {System.out.println("Token "+yytext()+" reconocido"); return new Symbol(sym.PAR_A,yyline,yycolumn,yytext());}
```

Inicia con la producción inicial que en este caso se usa la de defecto que se llama YYINITIAL indicándole en que orden se podrían detectar los tokens dentro de llave. Ejemplo:

```
7070
<YYINITIAL>{PAR_A}
```

Luego imprimimos el token y lo retornamos con el mismo nombre.

```
{System.out.println("Token "+yytext()+" reconocido"); return new Symbol(sym.PAR_A,yyline,yycolumn,yytext());}
{System.out.println("Token "+yytext()+" reconocido"); return new Symbol(sym.PAR_C,yyline,yycolumn,yytext());}
```

Luego de los símbolos declaramos las palabras reservadas y por ultimo el identificador para que no tome como identificador las reservadas. También se incluye un estado de STRING para que obtenga el token string.

```
<STRING> {
    \"          { yybegin(YYINITIAL); return new Symbol(sym.STRING, NuevoString.toString());}
    \\\n      { NuevoString.append('\\n'); }
    \\\n      { NuevoString.append(\"\\n\"); }
    \\\n      { NuevoString.append('\\n'); }
    \\\n      { NuevoString.append('\\n'); }
    \\\n      { NuevoString.append('\\n'); }
    {ENTER}   { yybegin(YYINITIAL);
               System.out.println("String sin finalizar." + yyline + yycolumn); }
    .         { NuevoString.append(yytext()); }
}
```

Por último declaramos un estado el cual será para errores léxicos.

```
<YYINITIAL>{ENTER}      { /* Ignorado */ }
<YYINITIAL> . {
    Errores nuevo=new Errores(TipoError.LEXICO,"Error con el token: "+yytext(),yyline+1,yycolumn+1);
    listaerrores.add(nuevo);
    String errLex = "Error léxico : '"+yytext()+"' en la línea: "+(yyline+1)+" y columna: "+(yycolumn+1);
    System.out.println(errLex);
}
<STRING> {
```

Para la parte de análisis sintactico se inicia con la declaración de métodos que son de ayuda para eliminar caracteres o crear objetos y con la parte de importaciones.

```

package AnalizadorA;
import java_cup.runtime.Symbol;
import java.util.LinkedList;
import Reportes.Errores;
import Reportes.Errores.TipoError;
import java.util.ArrayList;
import Expresion.*;
import Expresion.TipoExp.Tipos;
import AST.*;
import Instruccion.Instruccion;
import Instruccion.Print;
import Instruccion.AsignacionPosicion;
import Instruccion.DecFuncion;
import Objetos.Nulo;
import Operaciones.Aritmeticas;
import Operaciones.Operation;
import Operaciones.Logicas;
import Operaciones.Relacional;
import Instruccion.DecAsig;
import Expresion.Identificador;
import Operaciones.Ternarias;
import Operaciones.Unarias;
import Control.*;
import java.util.LinkedList;
parser code

```

```

public String remplazar(String t){
    t=t.replace("\n","\\n");
    t=t.replace("\t","\\t");
    t=t.replace("\\","\\\\");
    t=t.replace("\r","\\r");
    t=t.replace("\\\\","\\\\\\");
    return t;
}
public Nodo Generar(String s,LinkedList<Nodo>sent,int linea,int columna){
    return new DecFuncion(new LinkedList<Object>(),new Identificador(s,linea,columna),sent,linea,columna);
}
public Nodo Generar(String s,Nodo exp,LinkedList<Nodo>sent,int linea,int columna){
    if(exp instanceof Identificador){
        LinkedList<Object>parametros=new LinkedList<>();
        parametros.add(exp);
        return new DecFuncion(parametros,new Identificador(s,linea,columna),sent,linea,columna);
    }else{
        listaerrores.add(new Errores(TipoError.SINTACTICO,"Los parametros de una funcion solo pueden ser ID o ASIGNACION",linea,columna));
        return null;
    }
}

```

Luego se encuentran 2 metodos que son utilizados para la recuperacion de errores.

```

public LinkedList<Errores> listaErrores=new LinkedList<>();
public void syntax_error(Symbol s){
    Errores nuevo=new Errores(TipoError.SINTACTICO,"Error R de sintaxis "+s.value,s.left+1,s.right+1);
    listaerrores.add(nuevo);
    System.out.println("Error R de sintaxis: "+ s.value +" Linea "+(s.left+1)+" columna "+(s.right+1) );
}
public void unrecovered_syntax_error(Symbol s) throws java.Lang.Exception{
    Errores nuevo=new Errores(TipoError.SINTACTICO,"Error NR de sintaxis "+s.value,s.left+1,s.right+1);
    //Errores nuevo=new Errores(s.left+1,s.right+1,"Error NR de sintaxis "+s.value,Errores.Terror.SINTACTICO);
    listaerrores.add(nuevo);
    System.out.println("Error NR de sintaxis: "+ s.value +" Linea "+(s.left+1)+" columna "+(s.right+1) );
}

```

En esta seccion se deben declarar todos los tokens que son retornados por nuestro analisis lexico y nuestras producciones no terminales.

```

terminal String LLAV_A,LLAV_C,COR_A,COR_C,PAR_A,PAR_C;
terminal String OR,NOT,AND,MAYOR,MAYOR_I,MENOR,MENOR_I,DISTINTO,IGUAL_IGUAL,MAS,MENOS,DIV,POR,MODULO,POTENCIA,PREGUNTA,DOSPUNTOS,COMA;
terminal String ID,NUMERIC,STRING,UMENOS;
terminal String IGUAL,PYCOMA,FUNCTION;
terminal String INTEGER,NULO,TRUE,FALSE,IF,ELSE,SWITCH,BREAK,CONTINUE,CASE,DEFAULT,RETURN,WHILE,DO,FOR,IN;

non terminal AST INICIO;
non terminal Expresion EXP;
non terminal LinkedList<Nodo>CUERPO,CUERPOINTERNO;
non terminal LinkedList<Nodo> BLOQUE;
non terminal Nodo ASIGNACION,CONTROLES,CONTROL_IF,CONTROL_SWITCH,TRANSFERENCIAS,DECFUNCION,CONTROL_WHILE,CONTROL_DO_WHILE,CONTROL_FOR;
non terminal LinkedList<Instruccion>LIFS,LCASE;
non terminal Expresion ACCESOARREGLO;
non terminal LinkedList<Expresion> LCORCHETES,LISTAEXPRESION;
non terminal LinkedList<Object> LPAR;
non terminal Expresion LLAMADAS;
non terminal Nodo RETORNOS;

```

Esta es la parte mas importante si se desea utilizar una gramática ambigua ya que se declara la precedencia de los operadores para nuestras expresiones.

```
precedence right IGUAL;
precedence right PREGUNTA,DOSPUNTOS;
precedence left OR;
precedence left AND;
precedence left IGUAL_IGUAL,DISTINTO;
precedence nonassoc MENOR,MENOR_I,MAYOR_I,MAYOR;
precedence left MAS,MENOS;
precedence left POR,DIV,MODULO;
precedence left POTENCIA;
precedence right NOT;
precedence right UMENOS;
precedence left COR_A, COR_C;
//precedence left PAR_A, PAR_C;
```

La producción inicial se llama INICIO y retorna un nodo AST que será la raíz de nuestro árbol

```
start with INICIO;

INICIO ::= CUERPO: a { :AST arbol=new AST(a); ast=arbol; };
```

La producción Cuerpo retorna una lista de nodos

```
CUERPO ::= //ASIGNACION: a { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(a); RESULT=lista; }
//| CUERPO: c ASIGNACION: a { :c.add(a); RESULT=c; }
CONTROLES: c { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(c); RESULT=lista; }
| CUERPO: c CONTROLES: b { :c.add(b); RESULT=c; }
//| BLOQUE: b { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(new Bloque(b)); RESULT=lista; }
//| CUERPO: c BLOQUE: b { :c.add(new Bloque(b)); RESULT=c; }
TRANSFERENCIAS: a { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(a); RESULT=lista; }
| CUERPO: l TRANSFERENCIAS: a { :l.add(a); RESULT=l; }
DECFUNCION: d { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(d); RESULT=lista; }
| CUERPO: l DECFUNCION: d { :l.add(d); RESULT=l; }
LLAMADAS: l { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(l); RESULT=lista; }
| LLAMADAS: l PYCOMA { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(l); RESULT=lista; }
| CUERPO: l LLAMADAS: a PYCOMA { :l.add(a); RESULT=l; }
| CUERPO: l LLAMADAS: a { :l.add(a); RESULT=l; }
| CUERPO: l RETORNOS: r { :l.add(r); RESULT=l; }
RETORNOS: r { :LinkedList<Nodo> lista=new LinkedList<>(); lista.add(r); RESULT=lista; }
error PYCOMA { :RESULT=new LinkedList<>(); }
error PAR_C { :RESULT=new LinkedList<>(); }
error COR_C { :RESULT=new LinkedList<>(); }
;
```

La producción de asignación retorna un nodo DecAssign la cual es la que genera las variables.

La producción de expresión retorna un objeto expresión, se soluciona la ambigüedad con reglas de prioridad.

La producción de Bloque retorna una lista de sentencias

La producción de llamadas retorna un objeto de tipo llamada con el identificador de la función que se desea realizar.

```

;
LLAMADAS ::=
ID: i PAR_A PAR_C { :RESULT = new Llamadas(new Identificador(i, ileft+1, iright+1), new LinkedList<Expresion>(), new LinkedList<Expresion>()); }
| ID: i PAR_A LISTAEXPRESION: l PAR_C { :RESULT = new Llamadas(new Identificador(i, ileft+1, iright+1), l, new LinkedList<Expresion>()); }
| ID: i PAR_A PAR_C LCORCHETES: c { :RESULT = new Llamadas(new Identificador(i, ileft+1, iright+1), new LinkedList<Expresion>(), c); }
| ID: i PAR_A LISTAEXPRESION: l PAR_C LCORCHETES: c { :RESULT = new Llamadas(new Identificador(i, ileft+1, iright+1), l, c); }
;

```

La producción de CONTROL retorna el tipo de control que se desea realizar

```

*/
CONTROLES ::=
CONTROL_IF: a { :RESULT = a; }
| CONTROL_SWITCH: a { :RESULT = a; }
| CONTROL_DO_WHILE: a PYCOMA { :RESULT = a; }
| CONTROL_DO_WHILE: a { :RESULT = a; }
| CONTROL_WHILE: a { :RESULT = a; }
| CONTROL_FOR: a { :RESULT = a; }
;

```

La producción de control IF retorna un objeto de tipo IF que tiene una condición de tipo expresión y una lista de sentencias

```

;
CONTROL_IF ::=
IF: a PAR_A EXP: e PAR_C BLOQUE: b { :RESULT = new IF(e, b, null, aleft+1, aright+1); }
| IF: a PAR_A EXP: e PAR_C BLOQUE: b LIFS: l { :RESULT = new IF(e, b, l, aleft+1, aright+1); }
;

```

Si en esta producción vinieran mas else if o else se retorna una linkedlist de mas ifs .

```

LIFS ::=
ELSE: a IF PAR_A EXP: e PAR_C BLOQUE: b LIFS: l { l.addFirst(new ElseIf(e, b, aleft+1, aright+1)); :RESULT = l; }
| ELSE: a IF PAR_A EXP: e PAR_C BLOQUE: b { :LinkedList<Instruccion> a = new LinkedList<>(); a.add(new ElseIf(e, b, aleft+1, aright+1)); :RESULT = a; }
| ELSE: a BLOQUE: b { :LinkedList<Instruccion> a = new LinkedList<>(); a.add(new Else(b, aleft+1, asright+1)); :RESULT = a; }
;

```

Para la producción de CONTROL_SWITCH tiene la misma lógica que el IF la cual tiene un condición , una lista de case y un default si lo tiene.

```

CONTROL_SWITCH ::=
SWITCH: i PAR_A EXP: e PAR_C LLAV_A LCASE: l LLAV_C { :RESULT = new Switch(e, l, ileft+1, iright+1); }
| SWITCH: i PAR_A EXP: e PAR_C LLAV_A LLAV_C { :RESULT = new Switch(e, new LinkedList<Instruccion>(), ileft+1, iright+1); }
;

```

Para las producciones de ciclo tienen una lista de sentencias y un objeto de condición

```

CONTROL_WHILE ::=
WHILE: a PAR_A EXP: e PAR_C BLOQUE: b { :RESULT = new While(e, b, aleft+1, aright+1); }
;
CONTROL_DO_WHILE ::=
DO: a BLOQUE: b WHILE PAR_A EXP: e PAR_C { :RESULT = new DoWhile(e, b, aleft+1, aright+1); }
;
CONTROL_FOR ::=
FOR: a PAR_A ID: i IN EXP: e PAR_C BLOQUE: b { :RESULT = new For(new Identificador(i, ileft+1, iright+1), e, b, aleft+1, aright+1); }
;
LCASE ::=

```


Para las producciones de Accesos a las estructuras se viene una lista de Accesos.

```
ACCESOARREGLO:=  
  ID:a LCORCHETES:l{:RESULT=new Acceso(new Identificador(a,aleft+1,aright+1),l,aleft+1,aright+1);}  
  ;  
LCORCHETES:=  
  COR_A:a EXP:e COR_C{:LinkedList<Expresion>lista=new LinkedList<>();lista.addFirst(new AccesoUnico(e,aleft+1,aright+1));RESULT=lista;}  
  |COR_A:a EXP:e COR_C LCORCHETES:l{:l.addFirst(new AccesoUnico(e,aleft+1,aright+1));RESULT=l;}  
  |COR_A:a COR_A EXP:e COR_C COR_C{:LinkedList<Expresion>lista=new LinkedList<>();lista.addFirst(new AccesoDoble(e,aleft+1,aright+1));RESULT=lista;  
  |COR_A:a COR_A EXP:e COR_C COR_C LCORCHETES:l{:l.addFirst(new AccesoDoble(e,aleft+1,aright+1));RESULT=l;}  
  |COR_A:a EXP:f COMA EXP:c COR_C{:LinkedList<Expresion>lista=new LinkedList<>();lista.addFirst(new Acceso4(f,c,aleft+1,aright+1));RESULT=lista;  
  |COR_A:a EXP:f COMA EXP:c COR_C LCORCHETES:l{:l.addFirst(new Acceso4(f,c,aleft+1,aright+1));RESULT=l;}  
  |COR_A:a EXP:f COMA COR_C{:LinkedList<Expresion>lista=new LinkedList<>();lista.add(new Acceso4(f,null,aleft+1,aright+1));RESULT=lista;}  
  |COR_A:a EXP:f COMA COR_C LCORCHETES:l{:l.addFirst(new Acceso4(f,null,aleft+1,aright+1));RESULT=l;}  
  |COR_A:a COMA EXP:c COR_C{:LinkedList<Expresion>lista=new LinkedList<>();lista.add(new Acceso4(null,c,aleft+1,aleft+1));RESULT=lista;}  
  |COR_A:a COMA EXP:c COR_C LCORCHETES:l{:l.addFirst(new Acceso4(null,c,aleft+1,aright+1));RESULT=l;}  
  ;
```

Para la producción de declarar objetos se utiliza una producción especial en lo cual se llama a un método para validar si se puede declarar de esa forma ya que tiraba error de Reduce/Reduce al momento de declarar variables y funciones

```
DECLFUNCION:=  
  //ID:i IGUAL:g FUNCTION PAR_A LISTAPARAMETROS:p PAR_C BLOQUE:s{:RESULT=new DeclFuncion(p,new Identificador(i,ileft+1,iright+1),s,gleft+1,gright+1);}  
  //ID:i IGUAL:g FUNCTION PAR_A PAR_C BLOQUE:s{:RESULT=new DeclFuncion(new LinkedList<Object>(),new Identificador(i,ileft+1,iright+1),s,gleft+1,gright+1);}  
  //ID:i IGUAL:g PAR_A LISTAPARAMETROS:p PAR_C IGUAL MAYOR BLOQUE:s{:RESULT=new DeclFuncion(p,new Identificador(i,ileft+1,iright+1),s,gleft+1,gright+1);}  
  //ID:i IGUAL:g PAR_A PAR_C IGUAL MAYOR BLOQUE:s{:RESULT=new DeclFuncion(new LinkedList<Object>(),new Identificador(i,ileft+1,iright+1),s,gleft+1,gright+1);}  
  ID:a IGUAL PAR_A PAR_C IGUAL MAYOR BLOQUE:b{:RESULT=Generar(a,b,aleft+1,aright+1);}  
  |ID:a IGUAL EXP:e{:RESULT=new DecAsig(e,new Identificador(a,aleft+1,aright+1),aleft+1,aright+1);}  
  |ID:a IGUAL EXP:e PYCOMA{:RESULT=new DecAsig(e,new Identificador(a,aleft+1,aright+1),aleft+1,aright+1);}  
  |ID:a IGUAL PAR_A EXP:e PAR_C IGUAL MAYOR BLOQUE:b{:RESULT=Generar(a,e,b,aleft+1,aright+1);}  
  |ID:a IGUAL PAR_A EXP:e IGUAL:i EXP:e1 PAR_C IGUAL MAYOR BLOQUE:b{:RESULT=Generar(a,e,e1,b,ileft+1,iright+1,aleft+1,aright+1);}  
  |ID:a IGUAL PAR_A EXP:e COMA LPAR:l PAR_C IGUAL MAYOR BLOQUE:b{:RESULT=Generar(a,e,l,b,aleft+1,aright+1);}  
  |ID:a IGUAL PAR_A EXP:e IGUAL:i EXP:e1 COMA LPAR:l PAR_C IGUAL MAYOR BLOQUE:b{:RESULT=Generar(a,e,e1,l,b,ileft+1,iright+1,aleft+1,aright+1);}  
  |ID:a IGUAL FUNCTION PAR_A PAR_C BLOQUE:b{:RESULT=Generar(a,b,aleft+1,aright+1);}  
  |ID:a IGUAL FUNCTION PAR_A EXP:e PAR_C BLOQUE:b{:RESULT=Generar(a,e,b,aleft+1,aright+1);}  
  |ID:a IGUAL FUNCTION PAR_A EXP:e IGUAL:i EXP:e1 PAR_C BLOQUE:b{:RESULT=Generar(a,e,e1,b,ileft+1,iright+1,aleft+1,aright+1);}  
  |ID:a IGUAL FUNCTION PAR_A EXP:e COMA LPAR:l PAR_C BLOQUE:b{:RESULT=Generar(a,e,l,b,aleft+1,aright+1);}  
  |ID:a IGUAL FUNCTION PAR_A EXP:e IGUAL:i EXP:e1 COMA LPAR:l PAR_C BLOQUE:b{:RESULT=Generar(a,e,e1,l,b,ileft+1,iright+1,aleft+1,aright+1);}  
  |ACCESOARREGLO:a IGUAL:i EXP:e{:RESULT=new AsignacionPosicion(a,e,ileft+1,iright+1);}  
  |ACCESOARREGLO:a IGUAL:i EXP:e PYCOMA{:RESULT=new AsignacionPosicion(a,e,ileft+1,iright+1);}  
  ;  
LPAR:=  
  ID:a{:LinkedList<Object>lista=new LinkedList<>();lista.add(new Identificador(a,aleft+1,aright+1));RESULT=lista;}  
  |ID:a COMA LPAR:l{:l.addFirst(new Identificador(a,aleft+1,aright+1));RESULT=l;}  
  |ID:a IGUAL EXP:e{:LinkedList<Object>lista=new LinkedList<>();lista.add(new DecAsig(e,new Identificador(a,aleft+1,aright+1),aleft+1,aright+1));RESULT=lista;  
  |ID:a IGUAL EXP:e COMA LPAR:l{:l.addFirst(new Identificador(a,aleft+1,aright+1));RESULT=l;}  
  ;  
/*  
LISTAPARAMETROS:=
```

La producción de retornos que puede o no traer una expresión en respuesta.

```
RETORNOS:=  
  RETURN:r PAR_A EXP:e PAR_C{:RESULT=new Return(e,rleft+1,rright+1);}  
  |RETURN:r PAR_A EXP:e PAR_C PYCOMA{:RESULT=new Return(e,rleft+1,rright+1);}  
  |RETURN:r{:RESULT=new Return(new Nulo(rleft+1,rright+1),rleft+1,rright+1);}  
  |RETURN:r PYCOMA{:RESULT=new Return(new Nulo(rleft+1,rright+1),rleft+1,rright+1);}  
  ;
```

JavaCC

Cuenta con un área de importación de directivas.

```
package AnalizadorD;  
import java.util.LinkedList;  
import Reportes.Errores;  
import Reportes.Errores.TipoError;  
import java.util.ArrayList;  
import Expresion.*;  
import Expresion.TipoExp.Tipos;  
import AST.*;  
import Instruccion.Instruccion;  
import Instruccion.*;  
import Instruccion.Print;  
import Objetos.Nulo;  
import Operaciones.Aritmeticas;  
import Operaciones.Operation;  
import Operaciones.Logicas;  
import Operaciones.Relacional;  
import Instruccion.DecAsig;  
import Expresion.Identificador;  
import Operaciones.Ternarias;  
import Operaciones.Unarias;  
import Control.*;  
public class GeneradorC
```

Luego se declaran los métodos que van a servir para la ejecución de la aplicación como lo son para limpiar la cadena de caracteres que no se esperan

```

public class Gramatica{
    public String retirar(String t){
        t=t.replace("\\n","\\n");
        t=t.replace("\\t","\\t");
        t=t.replace("\\\\","\\");
        t=t.replace("\\r","\\r");
        t=t.replace("\\\\","\\");
        return t;
    }
}
PARSER_END(Gramatica)
/*

```

La siguiente parte del documento es la declaración de tokens léxicos.

```

*/
SKIP :{
    " "
    | "\t"
    | "\r"
    | "\n"
    | "<\"#\" (~[\"\\n\", \"\\r\"])*>"
    | "<\"#*\" (~[\"*\"])* \"*\" (\"*\" | ~[\"*\", \"#\"] (~[\"*\"])* \"*\")* \"#\">"
}

```

Esta parte son los tokens que no se desean ser procesados.

```

TOKEN : {
    <INTEGER: ([ "0"-"9" ])+>
    | <NUMERIC: ([ "0"-"9" ])+ "." ([ "0"-"9" ])+>
    | <PYCOMA: ";">
    | <MAS: "+">
    | <MENOS: "-">
    | <POR: "*">
    | <DIV: "/">
    | <COMA: ",">
    | <POTENCIA: "^">
    | <MODULO: "%">
    | <PAR_A: "(">
    | <PAR_C: ")">
    | <COR_A: "[">
    | <COR_C: "]">
    | <LLAV_A: "{">
    | <LLAV_C: "}">
    | <IGUAL: "=">
    | <DISTINTO: "!=">
    | <IGUAL_IGUAL: "==">
    | <MAYOR_I: ">=">
    | <MENOR_I: "<=">
    | <MAYOR: ">">
    | <IGUAL_MAYOR: "=>">
    | <MENOR: "<">
    | <AND: "&">
    | <OR: "|">
    | <NOT: "!">
    | <BOOLEANO: "true" | "false">
    | <IF: "if">
    | <ELSE: "else">
    | <SWITCH: "switch">
    | <DO: "do">

```

En la parte de tokens se declaran los símbolos y palabras reservadas del lenguaje.

```

MORE:{
    "\\\"" : STRING_STATE
}
<STRING_STATE> MORE:
{
    <~["\\\""]>
    | <("\\\" "\\\"")>
    | <("\\\" \"n\")>
    | <("\\\" "\\\"")>
    | <("\\\" \"r\")>
    | <("\\\" \"t\")>
}
<STRING_STATE> TOKEN:{
    <STRING:"\\\""> : DEFAULT
}

```

Los estados para el token de cadena

```

AST INICIO():
{
    AST arbol;
    LinkedList<Nodo>lista=new LinkedList();
    Nodo aux;
}
{
    (aux=CUERPO(){lista.add(aux);})+ <EOF>{arbol=new AST(lista);return arbol;}
}

Nodo CUERPO():
{
    Nodo aux;
}

```

Las producciones iniciales igual que en la otra gramática se busca hacer de la misma manera el árbol. La producción inicial se llama INICIO y retorna un árbol AST.

```

Nodo CUERPO():
{
    Nodo aux;
}
{
    try{
        (aux=CONTROLES()){return aux;}
        | (aux=DECLARACION()){return aux;}
        | (aux=RETORNOS()){return aux;}
    }catch(ParseException x){
        Token matchedToken = x.currentToken.next;
        Globales.VarGlobales.getInstance().AgregarEU(new Errores(Errores.TipoError.SINTACTICO,"Caracter no esperado "+matchedToken.image ,matchedToken.begin
        error_skyppto();
        return null;
    }
}
}

```

La producción Cuerpo regresa un nodo que puede ser instrucción, declaración o retorno.

```
*/  
void error_skypto():  
{  
    ParseException e=generateParseException();  
    Token t;  
}  
{  
    {  
        do{  
            t=getNextToken();  
            System.out.println("EERorr "+t.image);  
        }while(!(t.kind == PYCOMA || t.kind == LLAV_C||t.kind==EOF));  
    }  
}
```

Tenemos una producción para los errores.

```
LinkedList<Nodo>BLOQUE():  
{  
    LinkedList<Nodo>lista=new LinkedList();  
    Nodo aux=null;  
}  
{  
    <LLAV_A> (aux=CUERPOINTERNO()){lista.add(aux);})* <LLAV_C>{return lista;}  
}
```

Bloque nos regresa una lista de Nodos.

```
Nodo CUERPOINTERNO():  
{  
    Nodo aux;  
}  
{  
    try{  
        (aux=CONTROLES()){return aux;}  
        (aux=DECLARACION()){return aux;}  
        (aux=TRANSFERENCIAS()){return aux;}  
        (aux=RETORNOS()){return aux;}  
    }catch(ParseException x){  
        Token matchedToken = x.currentToken.next;  
        Globales.VarGlobales.getInstance().AgregarEU(new Errores(Errores.TipoError.SINTACTICO,"Caracter no esperado "+matchedToken.image ,matchedToken.beginLine));  
        error_skypto();  
        return null;  
    }  
}
```

Cuerpo interno nos regresa un objeto de tipo nodo.

En esta producción cuenta con todos los elementos declarables de la gramática debido a que no nos de conflicto todo se encuentra en esta producción ordenados de manera que las producciones con mas elementos estén arriba y las de menos abajo.

```

LOOKAHEAD(7) i=<IDENTIFICADOR><IGUAL> <PAR_A> (fun=LISTAPARAMETROS())? <PAR_C><IGUAL_MAYOR> DIO
| LOOKAHEAD(3) i=<IDENTIFICADOR> <IGUAL> e=EXP() (<PYCOMA>)? {return new DecAsig(e,new Identifica
| LOOKAHEAD(3) i=<IDENTIFICADOR><IGUAL> <FUNCTION> <PAR_A> (fun=LISTAPARAMETROS())? <PAR_C> blo=BL
| LOOKAHEAD(2) i=<IDENTIFICADOR> dim=LCORCHETES() <IGUAL> e=EXP() (<PYCOMA>)? {return new Asignac
| LOOKAHEAD(2) i=<IDENTIFICADOR><PAR_A>(par=LISTAEXP())? <PAR_C>(<PYCOMA>)? {return new Llamadas(

```

Cada uno tiene un LOOKAHEAD que nos indica cuantos caracteres tiene que esperar para hacer match con esa producción.

```

Nodo RETORNOS():
{
    Token t;
    Expresion ex;
}
{
    LOOKAHEAD(5) t=<RETORNO> <PAR_A> ex=EXP() <PAR_C> (<PYCOMA>)? {return new Return(ex,t.beginLine,t.beginColumn);}
    //|| LOOKAHEAD(3) t=<RETORNO> <PAR_A> ex=EXP() <PAR_C> {return new Return(ex,t.beginLine,t.beginColumn);}
    | LOOKAHEAD(2) t=<RETORNO> <PYCOMA> {return new Return(new Nulo(t.beginLine,t.beginColumn),t.beginLine,t.beginColumn);}
    | t=<RETORNO> {return new Return(new Nulo(t.beginLine,t.beginColumn),t.beginLine,t.beginColumn);}
}
}

```

Esta es la producción de retornos que nos devuelve un objeto de tipo return que puede tener una expresión o no.

```

LinkedList<Object> LISTAPARAMETROS():
{
    LinkedList<Object> lista=new LinkedList();
    Object a;
}
{
    a=PARAMETROS() {lista.add(a);} (<COMA> a=PARAMETROS() {lista.add(a);}) * {return lista;}
}
Object PARAMETROS():
{
    Token t;
    Expresion e;
}
{
    t=<IDENTIFICADOR> (<IGUAL> e=EXP() {return new DecAsig(e,new Identificador(t.image,t.beginLine,t.beginColumn),t.beginLine,t.beginColumn);})?
}

```

En la producción de lista de parámetros y parámetros se arma una linkedlist de expresiones o instrucciones por eso es de tipo object.

```

Expresion ACCESOS():
{
    Expresion e,d;
    Token t;
}
{
    LOOKAHEAD(100) t=<COR_A> e=EXP() <COR_C> {return new AccesoUnico(e,t.beginLine,t.beginColumn);}
    | LOOKAHEAD(120) t=<COR_A> e=EXP() <COMA> d=EXP() <COR_C> {return new Acceso4(e,d,t.beginLine,t.beginColumn);}
    | LOOKAHEAD(90) t=<COR_A> e=EXP() <COMA> <COR_C> {return new Acceso4(e,null,t.beginLine,t.beginColumn);}
    | LOOKAHEAD(2) t=<COR_A> <COR_A> e=EXP() <COR_C> <COR_C> {return new AccesoDoble(e,t.beginLine,t.beginColumn);}
    | LOOKAHEAD(2) t=<COR_A> <COMA> d=EXP() <COR_C> {return new Acceso4(null,d,t.beginLine,t.beginColumn);}
}

```

En la producción de ACCESOS se tiene un lookahead muy grande debido a que si vienen muchas expresiones se toma en cuenta el valor mas grande de tokens para hacer match.

```

Instruccion TRANSFERENCIAS():
{
    Token i;
}
{
    i=<BREAK> (<PYCOMA>)?{return new Break(i.beginLine,i.beginColumn);}
    |i=<CONTINUE>(<PYCOMA>)?{return new Continue(i.beginLine,i.beginColumn);}
}

Instruccion CONTROLES():

```

En la producción de transferencia solo se cuenta con dos producciones que son el break y continue para los ciclos.

```

Instruccion CONTROL_FOR():
{
    Token i,id;
    Expresion e;
    LinkedList<Nodo>blo;
}
{
    i=<FOR> <PAR_A> id=<IDENTIFICADOR> <IN> e=EXP() <PAR_C> blo=BLOQUE(){
}

```

Para la instrucción FOR se retorna un objeto de tipo for con una lista de instrucciones, un identificador de variable y una expresión para agregarle el valor.

Para las producciones de WHILE y DO_WHILE se tienen la misma cantidad de parámetros a diferencia del identificador.

```

Instruccion CONTROL_WHILE():
{
    Token i;
    Expresion e;
    LinkedList<Nodo>blo;
}
{
    i=<WHILE> <PAR_A> e=EXP() <PAR_C> blo=BLOQUE(){return new While(e,blo,i.beginLine,i.beginColumn);}
}

Instruccion CONTROL_DO_WHILE():
{
    Expresion e;
    Token i;
    LinkedList<Nodo>blo;
}
{
    i=<DO> blo=BLOQUE() <WHILE> <PAR_A> e=EXP() <PAR_C> (<PYCOMA>)? {return new Do_while(e,blo,i.beginLine,i.beginColumn);}
}

```


La instrucción SWITCH tiene un bloque de sentencias así como cada uno de sus hijos y tiene el comportamiento del IF a diferencia que si no encuentra un return sigue realizando las acciones

```
Instruccion CONTROL_SWITCH():
{
    Token i;
    Expresion e;
    LinkedList<Instruccion>Lcase=new LinkedList();
}

i=<SWITCH> <PAR_A> e=EXP() <PAR_C> <LLAV_A> (Lcase=LCASE())? <LLAV_C>{return new Switch(e,Lcase,i.beginLine,i.beginColumn);}

LinkedList<Instruccion>LCASE():
{
    Instruccion i;
    LinkedList<Instruccion>ins=new LinkedList();
}

(i=CASE(){ins.add(i);})+ (i=CDEFAULT(){ins.add(i);})? {return ins;}

Instruccion CASE():
{
    Token i;
    Expresion exp;
    LinkedList<Nodo>blo=new LinkedList();
    Nodo aux;
}

i=<CASE> exp=EXP() <DOSPUTOS> (aux=CUERPOINTERNO(){blo.add(aux);})* {return new Case(exp,blo,i.beginLine,i.beginColumn);}

Instruccion CDEFAULT():
```

El IF tiene un objeto if que es el primero que se va a evaluar y una lista de ELSE_IF o ELSE.

```
Instruccion CONTROL_IF():
{
    Token i,el,tel;
    Instruccion ins;
    LinkedList<Instruccion>Lifs=new LinkedList();
    Expresion condi;
    LinkedList<Nodo>blo,bloelse;
}

i=<IF><PAR_A> condi=EXP() <PAR_C> blo=BLOQUE()
(
    LOOKAHEAD(2) el=<ELSE> ins=CONTROL_IF()
    {
        LinkedList<Instruccion>aux=((IF)ins).getLifs();
        Expresion condicionNueva=((IF)ins).getCondicion();
        LinkedList<Nodo>bloqueelse=((IF)ins).getSentencias();
        ElseIf nuevo=new ElseIf(condicionNueva,bloqueelse,el.beginLine,el.beginColumn);
        Lifs.add(nuevo);
        for(Instruccion n : aux){
            Lifs.add(n);
        }
    }
    LOOKAHEAD(2) tel=<ELSE> bloelse=BLOQUE(){Lifs.add(new Else(bloelse,tel.beginLine,tel.beginColumn));}
)?
{return new IF(condi,blo,Lifs,i.beginLine,i.beginColumn);}
}
```

La lista de expresiones que cuenta con una o muchas expresiones separadas por coma, esta producción es recursiva por la derecha.

```
LinkedList<Expresion>LISTAEXP():
{
    LinkedList<Expresion>lista=new LinkedList();
    Expresion e;
    Token d;
}
{
    (e=EXP()){lista.add(e);} | d=<TDEFAULT>{lista.add(new Default(d.beginLine,d.beginColumn));} ) (
    {return lista;}
}
```

A diferencia de CUP este tipo de analizador no se le indica la precedencia entonces esta configurado conforme a las producciones.

La mas alta es la que tiene menos precedencia.

```
Expresion EXP():
{
    Expresion a,b,c;
    Token t;
}
{
    a=CondicionOR()(t=<PREGUNTA>b=EXP()<DOSPUNTOS>c=EXP()){return new Ternarias(a,b,c,t.beginLine,t.beginColumn);}
    )?
    {return a;}
}
Expresion CondicionOR():
```

Iniciamos con las ternarias y luego con la producción de OR.

```
Expresion CondicionOR():
{
    Expresion a,b;
    Token t;
}
{
    a=CondicionAnd()(t=<OR>b=CondicionAnd()){a=new Logicas(a,b,Operacion.Operador.OR,t.beginLine,t.beginColumn);}
    )*
    {return a;}
}
Expresion CondicionAnd():
```

Luego viene la condición AND.

```
Expresion CondicionAnd():
{
    Expresion a,b;
    Token t;
}
{
    a=ExpresionIgualdad()(t=<AND>b=ExpresionIgualdad()){a=new Logicas(a,b,Operacion.Operador.AND,t.beginLine,t.beginColumn);}
    )*
    {return a;}
}
```

En esta producción hace cambio a las expresiones relacionales.

```

Expresion ExpresionIgualdad():
{
    Expresion a,b;
    Token t;
}
{
    a=ExpresionRelacional()(t=<IGUAL_IGUAL> b=ExpresionRelacional(){a=new Relacional(a,b,Operacion.Operador.IGUAL_IGUAL,t.beginLine,t.beginColumn);}
    |t=<DISTINTO> b=ExpresionRelacional(){a=new Relacional(a,b,Operacion.Operador.DISTINTO,t.beginLine,t.beginColumn);}
    )*
    {return a;}
}

```

Iniciando por la de igualdad y luego como tienen las mismas precedencias se meten en una sola producción.

```

}
Expresion ExpresionRelacional():
{
    Expresion a,b;
    Token t;
}
{
    a=ExpresionAditiva()
    (
        t=<MAYOR> b=ExpresionAditiva(){a=new Relacional(a,b,Operacion.Operador.MAYOR,t.beginLine,t.beginColumn);}
        |t=<MENOR> b=ExpresionAditiva(){a=new Relacional(a,b,Operacion.Operador.MENOR,t.beginLine,t.beginColumn);}
        |t=<MAYOR_I>b=ExpresionAditiva(){a=new Relacional(a,b,Operacion.Operador.MAYOR_IGUAL,t.beginLine,t.beginColumn);}
        |t=<MENOR_I>b=ExpresionAditiva(){a=new Relacional(a,b,Operacion.Operador.MENOR_IGUAL,t.beginLine,t.beginColumn);}
    )*
    {return a;}
}
Expresion ExpresionAditiva():

```

Luego vienen las producciones aritmeticas.

```

}
Expresion ExpresionAditiva():
{
    Expresion a,b;
    Token t;
}
{
    a=ExpresionMultiplicativas()
    (
        t=<MAS>b=ExpresionMultiplicativas(){a=new Aritmeticas(a,b,Operacion.Operador.SUMA,t.beginLine,t.beginColumn);}
        |t=<MENOS>b=ExpresionMultiplicativas(){a=new Aritmeticas(a,b,Operacion.Operador.RESTA,t.beginLine,t.beginColumn);}
    )*
    {return a;}
}
Expresion ExpresionMultiplicativas():
{
    Expresion a,b;
    Token t;
}
{
    a=ExpresionUnaria()
    (
        t=<POR> b=ExpresionUnaria(){a=new Aritmeticas(a,b,Operacion.Operador.MULTIPLICACION,t.beginLine,t.beginColumn);}
        |t=<DIV>b=ExpresionUnaria(){a=new Aritmeticas(a,b,Operacion.Operador.DIVISION,t.beginLine,t.beginColumn);}
        |t=<POTENCIA>b=ExpresionUnaria(){a=new Aritmeticas(a,b,Operacion.Operador.POTENCIA,t.beginLine,t.beginColumn);}
        |t=<MODULO>b=ExpresionUnaria(){a=new Aritmeticas(a,b,Operacion.Operador.MODULO,t.beginLine,t.beginColumn);}
    )*
    {return a;}
}
Expresion ExpresionUnaria():

```

Por ultimo tenemos las producciones con mas precedencia y la EXP con un único valor.

```

}
Expresion ExpresionUnaria():
{
    Expresion exp,a;
    Token t;
}
{
    t=<MENOS>exp=ExpresionUnaria(){return new Unarias(exp,null,Operacion.Operador.RESTA,t.beginLine,t.beginColumn);}
    |t=<NOT>exp=ExpresionUnaria(){return new Unarias(exp,null,Operacion.Operador.NOT,t.beginLine,t.beginColumn);}
    |exp=Primitivo(){return exp;}
}
Expresion Primitivo():

```

```

}
Expresion Primitivo():
{
    Expresion aux;
    Token i;
    LinkedList<Expresion>dim=new LinkedList(),lex=new LinkedList();
}
{
    <NUMERIC>{return new Literal(Double.parseDouble(token.image),new TipoExp(Tipos.NUMERIC),token.beginLine,token.beginColumn);}
    |<BOOLEAN>{return new Literal(Boolean.parseBoolean(token.image),new TipoExp(Tipos.BOOLEAN),token.beginLine,token.beginColumn);}
    |<INTEGER>{return new Literal(Integer.parseInt(token.image),new TipoExp(Tipos.INTEGER),token.beginLine,token.beginColumn);}
    |<STRING>{return new Literal(retirar(token.image.substring(1,token.image.length()-1)),new TipoExp(Tipos.STRING),token.beginLine,token.beginColumn);}
    LOOKAHEAD(3)i=<IDENTIFICADOR> <PAR_A> <PAR_C> (dim-LCORCHETES())?{return new Llamadas(new Identificador(i.image,i.beginLine,i.beginColumn),lex,dim);}
    LOOKAHEAD(3)i=<IDENTIFICADOR> <PAR_A> lex-LISTAEXP() <PAR_C> (dim-LCORCHETES())?{return new Llamadas(new Identificador(i.image,i.beginLine,i.beginColumn),lex,dim);}
    LOOKAHEAD(2)i=<IDENTIFICADOR> dim-LCORCHETES(){return new Acceso(new Identificador(i.image,i.beginLine,i.beginColumn),dim,i.beginLine,i.beginColumn);}
    i=<IDENTIFICADOR>{return new Identificador(i.image,i.beginLine,i.beginColumn);}
    |<PAR_A> aux-EXP() <PAR_C>{return aux;}
    |<NULO>{return new Nulo(token.beginLine,token.beginColumn); }
}
}

```