```
package microjs.jcompiler.frontend.ast;

import java_cup.runtime.ComplexSymbolFactory.Location;
import microjs.jcompiler.middleend.kast.KAssign;
import microjs.jcompiler.utils.DotGraph;

public class Assign extends Statement {
    private String name;
    private Expr expr;

    public Assign(String name, Expr expr, Location startPos, Location endPos) {
        super(startPos, endPos);
        this.name = name;
                this.expr = expr;
    }

    @Override
    public KAssign expand() {
        return new KAssign(name, expr.expand(), getStartPos(), getEndPos());
    }

    @Override
    protected void prettyPrint(StringBuilder buf, int indent_level) {
        indent(buf, indent_level);
        buf.append(name);
        buf.append(" = ");
        expr.prettyPrint(buf);
    }

        @Override
        protected String buildDotGraph(DotGraph graph) {
                String assignNode = graph.addNode("Assign[" + name + "]");
                String exprNode = expr.buildDotGraph(graph);
                graph.addEdge(assignNode, exprNode, "expr");

                return assignNode;
        }



}
```

```java
package microjs.jcompiler.frontend.ast;

import java.util.List;

import java_cup.runtime.ComplexSymbolFactory.Location;
import microjs.jcompiler.middleend.kast.KIf;
import microjs.jcompiler.middleend.kast.KSeq;
import microjs.jcompiler.middleend.kast.KStatement;
import microjs.jcompiler.utils.DotGraph;

public class If extends Statement {
    private Expr cond;
    private List<Statement> thens;
    private List<Statement> elses;

    public If(Expr cond, List<Statement> thens, List<Statement> elses, Location
startPos, Location endPos) {
        super(startPos, endPos);
        this.cond = cond;
        this.thens = thens;
        this.elses = elses;
    }

    @Override
    public KIf expand() {
        // then part
        Location thenStartPos = getStartPos(); // XXX: good approximation ?
        Location thenEndPos = getStartPos();
        List<KStatement> kthens = Statement.expandStatements(thens);
        KStatement kthen = KSeq.buildKSeq(kthens, thenStartPos, thenEndPos);

        // else part
        Location elseStartPos = thenEndPos; // XXX: good approximation ?
        Location elseEndPos = thenEndPos;
        List<KStatement> kelses = Statement.expandStatements(elses);
        KStatement kelse = KSeq.buildKSeq(kelses, elseStartPos, elseEndPos);
        return new KIf(cond.expand(), kthen, kelse, getStartPos(), getEndPos());
    }

        @Override
        protected String buildDotGraph(DotGraph graph) {
                String ifNode = graph.addNode("If");
                String condNode = cond.buildDotGraph(graph);
                graph.addEdge(ifNode, condNode, "cond");
                String thenNode = cond.buildDotGraph(graph);
                graph.addEdge(ifNode, thenNode, "then");
                String elseNode = cond.buildDotGraph(graph);
                graph.addEdge(ifNode, elseNode, "else");

                return ifNode;
        }


    @Override
    protected void prettyPrint(StringBuilder buf, int indent_level) {
        indent(buf, indent_level);
        buf.append("if (");
        cond.prettyPrint(buf);
        buf.append(") {\n");
        Statement.prettyPrintStatements(buf, thens, indent_level + 1);
        indent(buf, indent_level);
        buf.append("} else {\n");
        Statement.prettyPrintStatements(buf, elses, indent_level + 1);
        indent(buf, indent_level);
        buf.append("}");
    }
```

```java
}
```

```
/* JFlex specification for JCompiler */

package microjs.jcompiler.frontend.lexer;

import java_cup.runtime.*;
import java_cup.runtime.ComplexSymbolFactory.Location;
import java_cup.runtime.ComplexSymbolFactory.ComplexSymbol;
import microjs.jcompiler.frontend.parser.sym;

/**
 * This class is a simple example lexer.
 */

%%

%class Lexer
%public
%unicode
%implements java_cup.runtime.Scanner
%function next_token
%type java_cup.runtime.Symbol
%line
%column

%{
  private ComplexSymbolFactory symbolFactory = new ComplexSymbolFactory();
  // StringBuffer string = new StringBuffer();

  private Symbol symbol(String name, int type) {
    return symbolFactory.newSymbol(name, type,
             new Location(yyline+1, yycolumn +1),
             new Location(yyline+1,yycolumn+yylength()));
  }
  private Symbol symbol(String name, int type, Object value) {
    return symbolFactory.newSymbol(name, type,
             new Location(yyline+1, yycolumn +1),
             new Location(yyline+1,yycolumn+yylength()), value);
  }
%}

Identifier = [a-zA-Z][a-zA-Z0-9]*

Digit = [0-9]





LineTerminator = ( \u000D\u000A
                       | [\u000A\u000B\u000C\u000D\u0085\u2028\u2029] )

%x COMMENTAIRE_C


%%


{LineTerminator} { /* ignore */ }

[ \t\f\n]+      { /* ignore */ }

{Digit}+        { return symbol("INT", sym.INT, Integer.parseInt(yytext())); }


var             { return symbol("VAR", sym.VAR); }
```

```
let             { return symbol("LET", sym.LET); }
true            { return symbol("BOOL", sym.BOOL, true); }
false           { return symbol("BOOL", sym.BOOL, false); }
if              { return symbol("IF", sym.IF); }
else            { return symbol("ELSE", sym.ELSE); }
function        { return symbol("FUNCTION", sym.FUNCTION); }
lambda          { return symbol("LAMBDA", sym.LAMBDA); }
return          { return symbol("RETURN", sym.RETURN); }

\;              { return symbol("SEMICOL", sym.SEMICOL); }
\,              { return symbol("COMMA", sym.COMMA); }
\=              { return symbol("EQ", sym.EQ); }
\{              { return symbol("LCURLY", sym.LCURLY); }
\}              { return symbol("RCURLY", sym.RCURLY); }
\(              { return symbol("LPAREN", sym.LPAREN); }
\)              { return symbol("RPAREN", sym.RPAREN); }
\+              { return symbol("PLUS", sym.PLUS); }
\-              { return symbol("MINUS", sym.MINUS); }
\*              { return symbol("TIMES", sym.TIMES); }
\/              { return symbol("DIV", sym.DIV); }
"=="            { return symbol("EQEQ", sym.EQEQ); }

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

\/\/.*\R        { /* ignore */ }        /* commentaire en ligne */

"/*"                      { yybegin(COMMENTAIRE_C); } /* commentaire C */
<COMMENTAIRE_C>[^*]+      { /* ignore */ }
<COMMENTAIRE_C>\*+        { /* ignore */ }
<COMMENTAIRE_C>\**"*/"    { yybegin(YYINITIAL); }


/* error fallback */
.                         {  // very strange "bug"
                            if (yytext() == "\\u000A") { /* ignore */
                               System.err.println(
                                 "WARNING: strange fallback character");
                            } else { throw new Error("Illegal character <"+
                                                       yytext()+">"); }

                          }

<<EOF>>         {return symbol("END", sym.END); }
```

```
package microjs.jcompiler.frontend.parser;

import java.util.List;
import java.util.LinkedList;

import java_cup.runtime.*;
import microjs.jcompiler.frontend.lexer.Lexer;
import microjs.jcompiler.frontend.ast.*;

terminal VAR, LET, EQ,
         LPAREN, RPAREN, LCURLY, RCURLY, /* LBRACKET, RBRACKET, */
         IF, ELSE,
         FUNCTION, LAMBDA, RETURN,
         EQEQ, PLUS, MINUS, TIMES, DIV,
         SEMICOL, COMMA;

terminal END;

terminal String IDENTIFIER;
terminal Integer INT;
terminal Boolean BOOL;

non terminal Prog       program;
non terminal Statement  statement;
non terminal Statement  opened_statement, closed_statement;
non terminal Expr       expr;
non terminal Statement  function;

non terminal List<Statement> statements;
non terminal List<Statement> block;
non terminal List<String>    parameters;
non terminal List<Expr>      arguments;

precedence left     EQEQ;
precedence left     PLUS, MINUS;
precedence left     TIMES, DIV;


program ::=
    END
        {: RESULT = new Prog("", new LinkedList<Statement>(), null, null); :}
  | statements:prog
        {: RESULT = new Prog("", prog, progxleft, progxright); :}
;


statements ::=          /***** pas de vide *****/
    statement:st
        {:
           LinkedList<Statement> tmp = new LinkedList<Statement>();
           if (st != null) {
              tmp.add(st);
           }
           RESULT = tmp;
        :}
 | statements:sts  statement:st
        {:
           if (st != null) {
              ((LinkedList<Statement>) sts).add(st);
           }
           RESULT = sts;
        :}
;


statement ::=
```

```
    SEMICOL
        {:
           RESULT = null;
        :}
  | opened_statement:ost  SEMICOL
        {:
           RESULT = ost;
        :}
  | closed_statement:cst
        {:
           RESULT = cst;
        :}
;


opened_statement ::=
    IDENTIFIER:id  EQ  expr:e
        {:
           RESULT = new Assign(id, e, idxleft, exright);
        :}
  | VAR:v  IDENTIFIER:var  EQ  expr:e
        {:
           RESULT = new Var(var, e, vxleft, exright);
        :}
  | LET:l  IDENTIFIER:var  EQ  expr:e
        {:
           RESULT = new Let(var, e, null, lxleft, exright);
        :}
  | expr:e
        {:
           RESULT = new VoidExpr(e, exleft, exright);
        :}
  | RETURN:r expr:e
        {:
           RESULT = new Return(e, rxleft, exright);
        :}
;


closed_statement ::=
    IF:i  LPAREN  expr:cond  RPAREN  block:thens
        {:
           RESULT = new If(cond,
                           thens,
                           new LinkedList<Statement>(),
                           ixleft, thensxright);
        :}
  | IF:i  LPAREN  expr:cond  RPAREN  block:thens  ELSE  block:elses
        {:
           RESULT = new If(cond, thens, elses, ixleft, elsesxright);
        :}
  | function:f
        {:
           RESULT = f;
        :}
;


function ::=
    FUNCTION:f  IDENTIFIER:id  LPAREN  RPAREN  block:body
        {:
           RESULT = new Function(id, new LinkedList<String>(),
                                 body, fxleft, bodyxright);
        :}
  | FUNCTION:f  IDENTIFIER:id  LPAREN  parameters:params  RPAREN  block:body
        {:
           RESULT = new Function(id, params, body, fxleft, bodyxright);
        :}
;


block ::=
    LCURLY  RCURLY
        {:
           RESULT = new LinkedList<Statement>();
        :}
  | LCURLY  statements:sts  RCURLY
        {:
           RESULT = sts;
        :}
;


parameters ::=          /***** pas de vide () ou de (...;;;...) *****/
    IDENTIFIER:id
        {:
           LinkedList<String> tempList = new LinkedList<String>();
           tempList.add(id);
           RESULT = tempList;
        :}
  | parameters:params  COMMA  IDENTIFIER:id
        {:
           ((LinkedList<String>)params).add(id);
           RESULT = params;
        :}
;


expr ::=
    INT:n
        {:
           RESULT = new IntConst(n, nxleft, nxright);
        :}
  | BOOL:b
        {:
           RESULT = new BoolConst(b, bxleft, bxright);
        :}
  | expr:fun  LPAREN:l  RPAREN:r
        {:
           RESULT = new Funcall(fun, new LinkedList<Expr>(),
                                funxleft, rxright);
        :}
  | expr:fun  LPAREN  arguments:args  RPAREN
        {:
           RESULT = new Funcall(fun, args, funxleft, argsxright);
        :}
  | LAMBDA:l  LPAREN  parameters:params  RPAREN  block:body
        {:
           RESULT = new Lambda(params, body, lxleft, bodyxright);
        :}
  | IDENTIFIER:var
        {:
           RESULT = new EVar(var, varxleft, varxright);
        :}
  | expr:l  PLUS  expr:r
        {:
           RESULT = new BinOp("+", l, r, lxleft, rxright);
        :}
  | expr:l  MINUS  expr:r
        {:
           RESULT = new BinOp("-", l, r, lxleft, rxright);
        :}
```

```
  | expr:l  TIMES  expr:r
        {:
           RESULT = new BinOp("*", l, r, lxleft, rxright);
        :}
  | expr:l  DIV  expr:r
        {:
           RESULT = new BinOp("/", l, r, lxleft, rxright);
        :}
  | expr:l  EQEQ  expr:r
        {:
           RESULT = new BinOp("==", l, r, lxleft, rxright);
        :}
  | LPAREN  expr:e  RPAREN
        {:
           RESULT = e;
        :}
;


arguments ::=           /***** pas de vide () ou de (...;;;...) *****/
    expr:e
        {:
           LinkedList<Expr> tempList = new LinkedList<Expr>();
           tempList.add(e);
           RESULT = tempList;
        :}
  | arguments:args  COMMA  expr:e
        {:
           ((LinkedList<Expr>)args).add(e);
           RESULT = args;
        :}
;
```