Appendix B

The Squeal Grammar

This is the grammar for Squeal in the format expected by the Java Compiler Compiler (JavaCC) [36].

```
options {
  IGNORE_CASE = true;
  MULTI = true;
}

PARSER_BEGIN(squealParser)
package squeal;
import SymbolTable;

public class squealParser {

  public static void main(String args[]) throws ParseError {
    squealParser parser = new squealParser(System.in);
    ASTstatement ast = parser.statement();
    ast.dump("");
  }

  public void reinit() { jjtree.reset(); }
}

PARSER_END(squealParser)

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

TOKEN :
{
        < SEMICOLON: ";">
|       < COLON: ":" >
|       < COUNT: "COUNT">
|       < SELECT: "SELECT">
|       < MSELECT: "MSELECT">
```

```
|       < UPDATE: "UPDATE">
|       < FROM: "FROM">
|       < ALL: "ALL">
|       < DISTINCT: "DISTINCT">
|       < UNIQUE: "UNIQUE">
|       < COMMA: "," >
|       < STAR: "*">
|       < PERIOD: "." >
|       < WHERE: "WHERE">
|       < GROUPBY: "GROUP BY">
|       < HAVING: "HAVING">
|       < LPAREN: "(" >
|       < RPAREN: ")" >
|       < NOT: "NOT">
|       < AND: "AND" | "&">
|       < OR: "OR" | "|">
|       < NEQ: "<>">
|       < LESS: "<">
|       < EQUALS: "=">
|       < GREATER: ">">
|       < GTE: ">=">
|       < LTE: "<=">
|       < LIKE: "LIKE">
|       < MATCHES: "MATCHES">
|       < BETWEEN: "BETWEEN">
|       < ORDERBY: "ORDER BY">
|       < SQUOTE: "'" >
|       < DQUOTE: "\"" >
|       < SSTRING: <SQUOTE> (~["'"])* <SQUOTE>>
|       < DSTRING: <DQUOTE> (~["\""])* <DQUOTE>>
|       < SLASH: "/">
|       < PLUS: "+">
|       < MINUS: "-">
|       < ASC: "ASC">
|       < DESC: "DESC">
|       < AVG: "AVG">
|       < MAX: "MAX">
|       < MIN: "MIN">
|       < SUM: "SUM">
|       < AS: "AS">
|       < DELETE: "DELETE">
|       < DROP: "DROP">
|       < QMARK: "?" >
|       < LET: "let" >
|       < DEFFUNC: "DEFFUNC">
|       < DEFPROC: "DEFPROC">
|       < ENDPROC: "ENDPROC">
|       < FETCH: "FETCH">
|       < PRINT: "PRINT">
|       < INTO: "INTO">
|       < HELP: "HELP">
|       < SET: "SET">
|       < IN: "IN">
|       < QUIT: "QUIT">
```

```
|       < EXIT: "EXIT">
|       < INSERT: "INSERT">
|       < VALUES: "VALUES">
|       < DESCRIBE: "DESCRIBE">
|       < UNLESS: "UNLESS">
|       < POUND: "#">
|       < NEW: "NEW">
|       < INPUT: "INPUT">
|       < OUTPUT: "OUTPUT">
|       < ELSE: "ELSE">
|       < CREATE: "CREATE">
|       < TABLE: "TABLE">
|       < CHAR: "CHAR">
|       < VARCHAR: "VARCHAR">
|       < BINARY: "BINARY">
|       < VARBINARY: "VARBINARY">
|       < INT: "INT">
|       < CONVERT: "CONVERT">
|       < VALUE_ID: "value_id">
|       < URL_ID: "url_id">
|       < ID: (<POUND>)*(<LETTER>)+(<LETTER>|<DIGIT>|<UNDERSCORE>)*>
|       < LETTER: ["A"-"Z", "a"-"z"]>
|       < UNDERSCORE: "_" >
|       < NUMBER: (<DIGIT>)+>
|       < DIGIT: ["0"-"9"]>
}


//// Statements

ASTstatement statement():{Token into = null;}
{
        <EOF> {return null;}
 |       nontrivialStatement() statement_separator() {return jjtThis;}
}


void nontrivialStatement()#void:{} {
  printStatement()
 |       letStatement()
 |       deffuncStatement()
 |       computeStatement()
 |       callStatement()
 |       defprocStatement()
 |  helpStatement()
 |       quitStatement()
 |       inputStatement()
 |       outputStatement()

        // SQL
 |       selectStatement(false)
 |       createStatement()
 |       dropStatement()
 |       deleteStatement()
```

```
 |       updateStatement()
 |       insertStatement()
 |       describeStatement()
}

void quitStatement():{}
{
        <QUIT> | <EXIT>
}

void helpStatement():{String s;}
{
        LOOKAHEAD(<HELP><LPAREN>)
        <HELP> <LPAREN> (s=identifier()) <RPAREN> {jjtThis.setName(s);}
|       <HELP> (s=identifier()) {jjtThis.setName(s);}
}

void statement_separator()#void:{}
{
        <SEMICOLON>
}

void selectStatement(boolean b):{Token cmd; String s=null; Token t = null;}
{
        (cmd=<SELECT>|cmd=<MSELECT>) (s=sel_restrict())? selectList()
        <FROM> tableList() restrict()
        ((t=<AS>|t=<INTO>) stringLiteral())?
         {jjtThis.setCommand(cmd.image);
          jjtThis.setInitialRestriction(s);
          if (t != null) jjtThis.setFileName();
          jjtThis.setIsSubSelect(b);}
}

String sel_restrict()#void:{}
{
        <ALL> {return "ALL";}
|       <DISTINCT> {return "DISTINCT";}
|       <UNIQUE> {return "UNIQUE";}
}

void subSelectStatement()#void:{}
{
        selectStatement(true)
}

void insertStatement():{Token id;}
{
        <INSERT> (<INTO>)? (id=<ID>) symbolList() insertStatementRHS()
         {jjtThis.setName(id.image);}
}

void insertStatementRHS()#void:{}
{
        <VALUES> argList()
```

```
|       subSelectStatement()
}

void updateStatement():{}
{
        <UPDATE> tableName() <SET> set_list() (whereDef())?
}

void describeStatement():{Token t;}
{
        <DESCRIBE> (t=<ID>) {jjtThis.setName(t.image);}
}

void deleteStatement():{Token t;}
{
        <DELETE> <FROM> (t=<ID>) (<WHERE> condition())?
         {jjtThis.setName(t.image);}
}

void dropStatement():{Token name;}
{
        <DROP> <TABLE> (name = <ID>)
         {jjtThis.setName(name.image);}
}

void createStatement():{Token name;}
{
        <CREATE> <TABLE> (name = <ID>) <LPAREN> columnDefList() <RPAREN>
         {jjtThis.setName(name.image);}
}

void columnDefList()#void:{}
{
        columnDef() (<COMMA> columnDef())*
}

void columnDef():{String name; String type;}
{
        (name = identifier()) (type=columnTypeDef())
         {jjtThis.setName(name); jjtThis.setType(type);}
}

String columnTypeDef()#void:{Token base, count;}
{
        (base=<INT>)
         {return base.image;}
|       (base=<URL_ID>)
         {return base.image;}
|       (base=<VALUE_ID>)
         {return base.image;}
|       (base=<CHAR>) <LPAREN> (count=<NUMBER>) <RPAREN>
         {return base.image + "(" + count.image + ")";}
|       (base=<VARCHAR>) <LPAREN> (count=<NUMBER>) <RPAREN>
         {return base.image + "(" + count.image + ")";}
```

```
|       (base=<VARBINARY>) <LPAREN> (count=<NUMBER>) <RPAREN>
        {return base.image + "(" + count.image + ")";}
|       (base=<BINARY>) <LPAREN> (count=<NUMBER>) <RPAREN>
        {return base.image + "(" + count.image + ")";}
}


//// Support for SELECT and FETCH statements

void selectList():{}
{
        selectItem() (<COMMA> selectItem())*
}

void selectItem():{String s=null;}
{
        expression() (<AS> (s=identifier()))?
        { if (s != null) jjtThis.setAlias(s); }
}

// Cells are allowed to handle transformed select statements
void namedArgument():{Token t=null;}
{
        (<NEW>)? symbolLiteral() (<PERIOD> symbolLiteral())?
        ((t=<EQUALS>)|(t=<LIKE>)) expression()
        { jjtThis.setOperator(t.image);}
}

void namedArgumentList():{}
{
        namedArgument() (<COMMA> namedArgument())*
}

void set_list()#void:{}
{
        namedArgumentList()
}

void computeList()#void:{}
{
        (namedArgumentList())?
}

void tableList():{}
{
        tableName() (<COMMA> tableName())*
}

void tableName():{String s; Token t2=null;}
{
        (s=identifier()) (t2=<ID>)? {jjtThis.setName(s);
          if (t2 != null) jjtThis.setAlias(t2.image);}
}
```

```
void restrict()#void:{}
{
        (whereDef())? (groupbyDef())? restrictEnd()
}

void whereDef():{}
{
        <WHERE> condition()
}


void restrictEnd()#void:{} {
        LOOKAHEAD(<HAVING>) havingDef() (orderbyDef())?
}

void groupbyDef():{}
{
        <GROUPBY> columnsList()
}

void havingDef():{}
{
        <HAVING> condition()
}

void orderbyDef():{}
{
        <ORDERBY> orderList()
}

void orderList():{}
{
        orderItem() (<COMMA> orderItem())*
}

void orderItem():{String s=null;}
{
        expression() (s = order_list_modifier())?
         { jjtThis.setModifier(s);}
}

String order_list_modifier()#void:{}
{
        <ASC> {return "ASC";}
|       <DESC> {return "DESC";}
}


void condition()#void:{}
{
        logicExpression()
}

void searchExpression()#void:{}
```

```
{
        logicExpression()
}

void columnsList():{}
{
        column() (<COMMA> column())*
}

void column():{Token t1; String s;}
{
        LOOKAHEAD(<ID> <PERIOD>) (t1=<ID>) <PERIOD> (s=identifier())
         { jjtThis.setTableName(t1.image);
           jjtThis.setColumnName(s);}
|       (s=identifier())
         { jjtThis.setColumnName(s);}
}

void convertExpression():{String def;}
{
        <CONVERT> <LPAREN> (def=columnTypeDef()) <COMMA> expression() <RPAREN>
         {jjtThis.setType(def);}
}


void aggregateExpression():{String s, r=null;}
{
        (s=aggregate()) <LPAREN> (r=agg_restrict())? expression() <RPAREN>
         {jjtThis.setName(s); jjtThis.setRestriction(r);}
}

void star():{}
{
        <STAR>
}

String aggregate()#void:{}
{
        <AVG> {return "AVG";}
|       <MAX> {return "MAX";}
|       <MIN> {return "MIN";}
|       <SUM> {return "SUM";}
|       <COUNT> {return "COUNT";}
}

String count()#void:{}
{
        <COUNT> {return "COUNT";}
}

String agg_restrict()#void:{}
{
        <ALL> {return "ALL";}
|       <DISTINCT> {return "DISTINCT";}
```

```
|       <UNIQUE> {return "UNIQUE";}
}

void inputStatement() :{}
{
        <INPUT> expression()
}

void outputStatement() :{}
{
        <OUTPUT> stringLiteral()
}

void printStatement() :{}
{
        (<QMARK>|<PRINT>) expression()
}

void letStatement():{Token t;}
{
        <LET> (t=<ID>) <EQUALS> expression() (<ELSE> expression())?
        {jjtThis.setName(t.image);}
}

void deffuncStatement():{Token t;}
{
        <DEFFUNC> (t=<ID>) symbolList() expression()
         {jjtThis.setName(t.image);}
}

void symbolList():{}
{
        <LPAREN> (symbolLiteral() (<COMMA> symbolLiteral())*)? <RPAREN>
}

void symbolLiteral():{String s;}
{
        (s=identifier()) { jjtThis.setName(s);}
}

void defprocStatement():{Token t;}
{
        <DEFPROC> (t=<ID>) symbolList()
        statement() (statement())*
        <ENDPROC>
         {jjtThis.setName(t.image);}
}

void callStatement()#void:{}
{
        funcall()
}

void computeStatement():{Token t; Token intoName = null; Token unless = null;}
```

```
{
        <FETCH> (t=<ID>) <LPAREN> computeList() <RPAREN>
        (<FROM> tableList() restrict())?
        (<INTO> (intoName = <ID>))?
        ((unless=<UNLESS>) condition())?
         {jjtThis.setName(t.image);
         if (unless != null)
           jjtThis.setUnlessClause();
         if (intoName != null)
           jjtThis.setIntoName(intoName.image);
         else
           jjtThis.setIntoName(t.image);}
}

void expression()#void :{}
{
   computedExpression()
}

void computedExpression():{Token t = null;}
{
        logicExpression()
}

void logicExpression()#void :{}
{
        disjunctionExpression()
}

void disjunctionExpression():{Token op = null;}
{
        conjunctionExpression() ((op=<OR>) disjunctionExpression())?
         {if (op != null) jjtThis.setOp(op.image);}
}

void conjunctionExpression():{Token op = null;}
{
        negationExpression() ((op=<AND>) conjunctionExpression())?
         {if (op != null) jjtThis.setOp(op.image);}
}

void negationExpression():{Token op = null;}
{
        ((op=<NOT>))? relExpression()
         {if (op != null) jjtThis.setOp(op.image);}
}

void relExpression():{String op = null;}
{
        sumExpression() ((op = rel_op()) sumExpression())?
         {jjtThis.setOp(op);}
}

String identifier()#void:{Token t;}
```

```
{
        ((t=<ID>)|(t=<URL_ID>)|(t=<VALUE_ID>)) {return t.image;}
}


String rel_op()#void:{}
{
        <LESS> {return("<");}
|       <EQUALS> {return("=");}
|       <GREATER> {return(">");}
|       <NEQ> {return("<>");}
|       <GTE> {return(">=");}
|       <LTE> {return("<=");}

|       LOOKAHEAD(<NOT> <IN>) <NOT> <IN> {return "NOT IN";}
|       <LIKE> {return "LIKE";}
|       <NOT> <LIKE> {return "NOT LIKE";}
|       <IN> {return "IN";}
}

void sumExpression():{String op = null;}
{
        productExpression() ((op=sum_op()) sumExpression())?
         {jjtThis.setOp(op);}
}

String sum_op()#void:{}
{
        <PLUS> {return("+");}
|       <MINUS> {return("-");}
}

void productExpression():{String op=null;}
{
        unaryExpression() ((op=product_op()) productExpression())?
         {jjtThis.setOp(op);}
}

String product_op()#void:{}
{
        <STAR> {return "*";}
|       <SLASH> {return "/";}
}

void unaryExpression():{}
{
        parenthesizedExpression()
|       <MINUS> parenthesizedExpression() {jjtThis.setNegated();}
}

void parenthesizedExpression():{}
{
        LOOKAHEAD(<LPAREN><SELECT>) <LPAREN> subSelectStatement() <RPAREN>
|       LOOKAHEAD(<LPAREN><MSELECT>) <LPAREN> subSelectStatement() <RPAREN>
```

```
|       <LPAREN> computedExpression() <RPAREN> {jjtThis.setParenthesized();}
|   LOOKAHEAD(identifier() <LPAREN>) funcall()
|       LOOKAHEAD(<ID> <PERIOD>) cell()
|       literal()
|       variable()
|       aggregateExpression()
|       star()
|       convertExpression()
}

void variable():{String s;}
{
        (s=identifier()) {jjtThis.setName(s);}
}

void literal()#void:{Token t;}
{
        numericLiteral()
  |     stringLiteral()
}

void numericLiteral():{Token t;}
{
        t=<NUMBER> {jjtThis.setNumber(t.image);}
}

void stringLiteral():{Token t;}
{
  t=<SSTRING> {jjtThis.setName(t.image.substring(1,
             t.image.length()-1));}
  |     t=<DSTRING> {jjtThis.setName(t.image.substring(1,
             t.image.length()-1));}
}

void funcall() :{String s;}
{
   (s = identifier()) argList()  {jjtThis.setName(s);}
}

void argList() : {}
{
        <LPAREN> (expression() (<COMMA> expression())*)? <RPAREN>
}

void cell():{ Token r, c;}
{
        (r=<ID>) <PERIOD> ((c=<ID>)|(c=<VALUE_ID>)|(c=<URL_ID>)|(c=<STAR>))
{jjtThis.setRelAndColName(r.image,c.image);}
}
```