```
Expr
  BinExpr
    Add
    BAnd
    BOr
    BXor
    Div
    Eql
    Gte
    Gt
    LAnd
    LNot
    LOr
    Lte
    Lt
    Mul
    Neq
    Sub
  Assign
  Id
  IntLit
  UnExpr
    Bnot
    UMinus
    UPlus
Stmt
  Block
  Empty
  ExprStmt
  If
  Print
  While
Stmts
```

```
$ jacc mini/Mini.jacc && jflex mini/Mini.jflex && javac mini/*java              WARNING: +
conflicts: 1 shift/reduce, 0 reduce/reduce

Reading "mini/Mini.jflex"
Constructing NFA : 167 states in NFA
Converting NFA to DFA :
............................................................
71 states before minimization, 59 states in minimized DFA
Old file "mini/MiniLexer.java" saved as "mini/MiniLexer.java~"
Writing code to "mini/MiniLexer.java"
$ cat variable.mini


int f;
int g = 1;
h = 2;
f = 4;
$ java mini.DotTest variable.mini > graph.dotty
$ cat iferror.mini
if (i > 0) { print i; } else { }
$ java mini.DotTest iferror.mini > graph.dotty
ERROR: syntax error

ERROR: Exception: java.lang.NullPointerException

$ cat sample.mini

i = 0;
j = 0;
while (i < 10) {
  i = i + 1;
  j = j + i;
}
print i;

print 1 + 2 * 3 + 4;

$ java mini.DotTest sample.mini > graph.dotty
$
```

```
%package mini
%extends Phase

%{
import compiler.*;
import java.io.*;
%}

%semantic Object

%token  WHILE IF ELSE PRINT
%token  '(' ')' '{' '}' ';'

%right  '='
%left   LOR
%left   LAND
%left   '|'
%left   '^'
%left   '&'
%left   EQL NEQ
%left   '<' '>' LTE GTE
%left   '+' '-'
%left   '*' '/'
%right  '!' '~' UMINUS UPLUS

%token  <TInt>   TINT
%token  <TBool>  TBOOL
%token  <TDoub>  TDOUB
%token  <TArray> TARRAY
%token  <Id>     IDENT
%token  <IntLit> INTLIT
%type   <Stmts>  stmts
%type   <Stmt>   stmt
%type   <Expr>   expr test
%type   <VType>  vtype

// TODO: this may be wrong. It's needed to give vintro a type for the second
// argument, which is really an Assign object.
%type   <Assign> vintro

%%
prog    : stmts                         { program = $1; }
        ;
stmts   : stmts stmt                    { $$ = $1.addStmt($2); }
        | stmt                          { $$ = new Stmts($1); }
        ;
stmt    : /* empty */ ';'               { $$ = new Empty(); }
        | expr ';'                      { $$ = new ExprStmt($1); }
        | WHILE test stmt               { $$ = new While($2, $3); }
        | IF test stmt ELSE stmt        { $$ = new If($2, $3, $5); }
        | IF test stmt                  { $$ = new If($2, $3, new Empty()); }
        | PRINT expr ';'                { $$ = new Print($2); }
        // | VTYPE '[' ']' IDENT                 { $$ = new VarArray($1, $4); }
        //| vtype vintro ';'            { $$ = new Variable($1, $2); }
        | vtype vintro ';'              { $$ = new Variable($1, $2); }
        | '{' stmts '}'                 { $$ = new Block($2); }
        ;
test    : '(' expr ')'                  { $$ = $2; }
        ;
expr    : '-' expr       %prec UMINUS   { $$ = new UMinus($2); }
        | '+' expr       %prec UPLUS    { $$ = new UPlus($2); }
        | '!' expr                      { $$ = new LNot($2); }
        | '~' expr                      { $$ = new BNot($2); }
```

```
        | expr '+' expr                { $$ = new Add($1, $3); }
        | expr '-' expr                { $$ = new Sub($1, $3); }
        | expr '*' expr                { $$ = new Mul($1, $3); }
        | expr '/' expr                { $$ = new Div($1, $3); }
        | expr '<' expr                { $$ = new Lt($1, $3); }
        | expr '>' expr                { $$ = new Gt($1, $3); }
        | expr LTE expr                { $$ = new Lte($1, $3); }
        | expr GTE expr                { $$ = new Gte($1, $3); }
        | expr NEQ expr                { $$ = new Neq($1, $3); }
        | expr EQL expr                { $$ = new Eql($1, $3); }
        | expr '&' expr                { $$ = new BAnd($1, $3); }
        | expr '|' expr                { $$ = new BOr($1, $3); }
        | expr '^' expr                { $$ = new BXor($1, $3); }
        | expr LAND expr               { $$ = new LAnd($1, $3); }
        | expr LOR  expr               { $$ = new LOr($1, $3); }
        // Note the following line now appears both as an expression
        // and as a variable introduction. This is needed to support
        // the assignment of an existing variable and also to assign
        // a value to a new variable. I'm sure it could be done better.
        | IDENT '=' expr               { $$ = new Assign($1, $3); }
        | IDENT                        { $$ = $1; }
        | INTLIT                       { $$ = $1; }
        ;
vintro  : IDENT '=' expr               { $$ = new Assign($1, $3); }
                                       // if no assignment is made, just
                                       // give it a default of "0".
        | IDENT                        { $$ = new Assign($1, new IntLit("0")); }
        ;
vtype   : TBOOL                        { $$ = new TBoolean(); }
        | TINT                         { $$ = new TInt(); }
        | TDOUB                        { $$ = new TDouble(); }
        | TARRAY                       { $$ = new TArray(); }
        ;


%%

    private MiniLexer lexer;
    private Stmts     program;

    public MiniParser(Handler handler, MiniLexer lexer) {
      super(handler);
      this.lexer = lexer;
      lexer.nextToken();
      parse();
    }

    public Stmts getProgram() {
      return program;
    }

    private void yyerror(String msg) {
      report(new Failure(msg));
    }
```

```
package mini;
import compiler.*;
import java.io.*;

%%

%class      MiniLexer
%public
%extends    Phase
%implements MiniTokens
%ctorarg    Handler handler
%init{
  super(handler);
%init}

%function   yylex
%int

%eofval{
  return ENDINPUT;
%eofval}

%{

  private int  token;

  private Expr semantic;

  public int getToken() {
    return token;
  }

  public Expr getSemantic() {
    return semantic;
  }

  public int nextToken() {
    try {
      semantic = null;
      token    = yylex();
    } catch (java.io.IOException e) {
      System.out.println("IO Exception occurred:\n" + e);
    }
    return token;
  }

%}

Identifier = [:jletter:] [:jletterdigit:]*

LineTerminator = \r|\n|\r\n
WhiteSpace = {LineTerminator} | [ \t\f]
InputCharacter = [^\r\n]

Comment            = {TraditionalComment} | {EndOfLineComment}
TraditionalComment = "/*" [^*] ~"*/" | "/*" "*"+ "/"
EndOfLineComment   = "//" {InputCharacter}* {LineTerminator}

%%

"("             { return '('; }
")"             { return ')'; }
"{"             { return '{'; }
```

```
"}"             { return '}'; }
";"             { return ';'; }

"="             { return '='; }
"=="            { return EQL; }
">"             { return '>'; }
">="            { return GTE; }
"<"             { return '<'; }
"<="            { return LTE; }
"!"             { return '!'; }
"~"             { return '~'; }
"!="            { return NEQ; }
"&"             { return '&'; }
"&&"            { return LAND; }
"|"             { return '|'; }
"||"            { return LOR; }
"^"             { return '^'; }
"*"             { return '*'; }
"+"             { return '+'; }
"-"             { return '-'; }
"/"             { return '/'; }

"while"         { return WHILE; }
"if"            { return IF; }
"else"          { return ELSE; }
"print"         { return PRINT; }

"int"           { return TINT; }
"boolean"       { return TBOOL; }
//"double"        { return TDOUB; }

{Identifier}    { semantic = new Id(yytext());     return IDENT; }

[0-9]+          { semantic = new IntLit(yytext()); return INTLIT; }

{WhiteSpace}    { /* ignore */ }
{Comment}       { /* ignore */ }

.|\n            { System.out.println("Invalid input");
                  System.exit(1);
                }
```

```
package mini;


/** Abstract syntax for an Integer type.
 */
class TInt extends VType {
    TInt() {
      // do nothing.
    }

    // Simply output our type, since we don't have children.
    public void indent(IndentOutput out, int n) {
        out.indent(n, "Integer");
    }

    // same with dot output.
    public int toDot(DotOutput dot, int n) {
        return dot.node("Integer", n);
    }

    /** Return a string that provides a simple description of this
     *  particular type of operator node.
     */
    String label() { return "integer"; }
}
```

```
package mini;


/** Abstract syntax for variable types.
 */
public abstract class VType {

    /** Print an indented description of this abstract syntax node,
     *  including a name for the node itself at the specified level
     *  of indentation, plus more deeply indented descriptions of
     *  any child nodes.
     */
    public abstract void indent(IndentOutput out, int n);

    /** Output a description of this node (with id n), including a
     *  link to its parent node (with id p) and returning the next
     *  available node id.
     */
    public int toDot(DotOutput dot, int p, int n) {
        dot.join(p, n);
        return toDot(dot, n);
    }

    /** Output a description of this node (with id n), returning the
     *  next available node id after this node and all of its children
     *  have been output.
     */
    public abstract int toDot(DotOutput dot, int n);
}
```

```
package mini;


/** Abstract syntax for variable types.
 */
class VarType {
    String type;
    VarType(String name) {
        this.type = type;
    }

    /** Print an indented description of this abstract syntax node,
     *  including a name for the node itself at the specified level
     *  of indentation, plus more deeply indented descriptions of
     *  any child nodes.
     */
    public void indent(IndentOutput out, int n) {
        //type.indent(n, "VType, " + type);
    }

    /** Output a description of this node (with id n), returning the
     *  next available node id after this node and all of its children
     *  have been output.
     */
    public int toDot(DotOutput dot, int n) {
        return dot.node("VType, " + type, n);
    }
}
```

```
package mini;


/** Abstract syntax for variable statements.
 */
class Variable extends Stmt {
    private VType type;
    private Assign assign;

    Variable(VType type, Assign assign) {
        this.type = type;
        this.assign = assign;
    }

    /** Print an indented description of this abstract syntax node,
     *  including a name for the node itself at the specified level
     *  of indentation, plus more deeply indented descriptions of
     *  any child nodes.
     */
    public void indent(IndentOutput out, int n) {
        out.indent(n, "variable");
        type.indent(out, n+1);
        assign.indent(out, n+1);
    }

    /** Output a description of this node (with id n), returning the
     *  next available node id after this node and all of its children
     *  have been output.
     */
    public int toDot(DotOutput dot, int n) {
        return assign.toDot(dot, n,
               type.toDot(dot, n,
               dot.node("variable", n)));
    }
}
```