

*Rule:* **<declaration> -> <var> <iden> <colon> <type> <initial\_value> <semi>**

*e.g.* **var num2: integer = 2020;**

class Declaration {

*String* iden;

*String* type;

Initial\_value init\_value;

public Declaration(Symbol lhs) {

iden = lhs.children.get(1).lexeme;

type = lhs.children.get(3).lexeme;

init\_value = new Initial\_value(lhs.children.get(4));

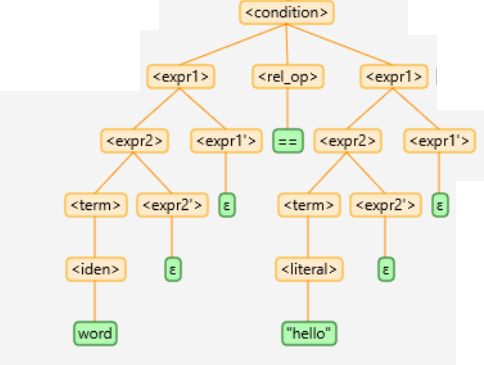
}

public void interpret() {

Variable x = new Variable(iden, type, -1);

}

}



*Rule:* **<condition> → <expr1> <rel\_op> <expr1>**

*e.g.* **word == “hello”**

class Condition {

Expr1 first;

*String* rel\_op;

Expr1 second;

public Condition(Symbol lhs) {

first = new Expr1(lhs.children.get(0));

rel\_op = lhs.children.get(1).lexeme;

second = new Expr1(lhs.children.get(2));

}

public boolean interpret() {

switch(rel\_op) {

case “==”: return first.interpret() == second.interpret();

case “!=”: return first.interpret() != second.interpret();

case “<=”: return first.interpret() <= second.interpret();

case “>=”: return first.interpret() >= second.interpret();

case “<”: return first.interpret() < second.interpret();

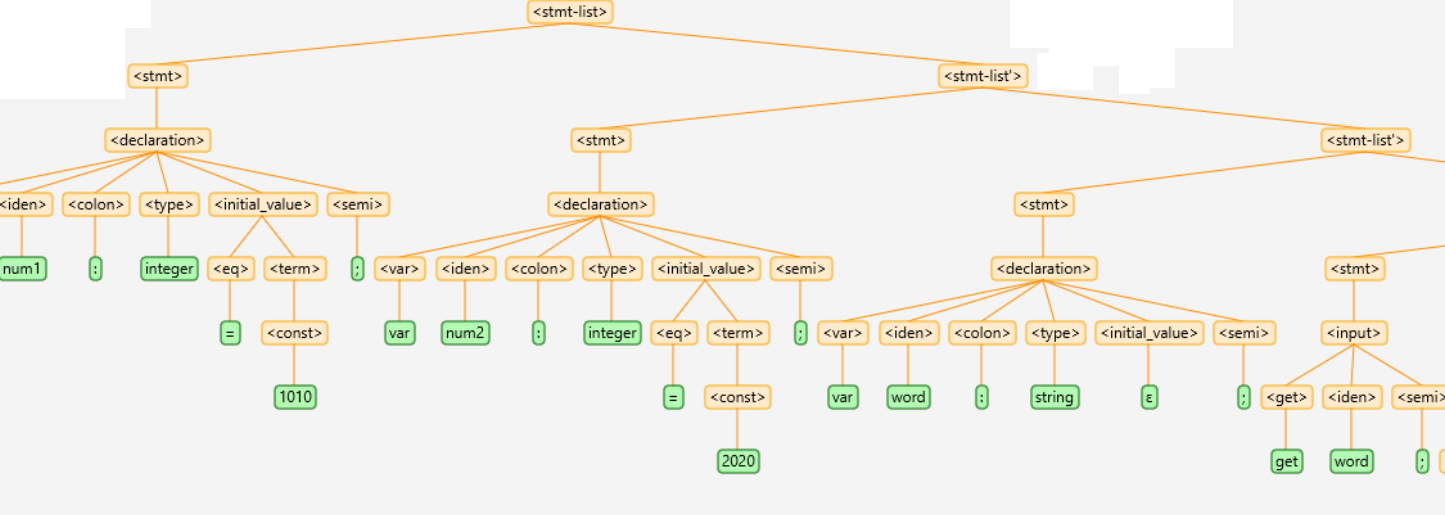
case “>”: return first.interpret() > second.interpret();

}

return false;

}

}

****

*Rule:* **<stmt-list> → <stmt> <stmt-list'>**

*e.g.* **var num1 : integer = 1010; // <-- <stmt>** (for the first statement)

**var num2 : integer = 2020; // <-- <stmt\_list’>** (for all succeeding statements after <stmt>)

**var word : string;**

**get word;**

class Stmt\_list {

Stmt firstStatement;

Stmt\_list\_prime succeedingStatements;

public Stmt\_list(Symbol lhs) {

firstStatement = new Stmt(lhs.children.get(0));

succeedingStatements = new Stmt\_list\_primt(lhs.children.get(1));

}

public void interpret() {

firstStatement.interpret();

succeedingStatements.interpret();

}

}

Sample **interpret()** method for EXPRESSION class to evaluate expressions with different types of operands

class EXPR {

LinkedList<Symbol> expression;

java.util.Deque<***Object***> operands = new LinkedList();

public EXPR\_1(Symbol lhs) {

expression = Expression.getExpression(lhs);

}

public ***Object*** interpret() {

Variable var;

int result = 0;

while (!expression.isEmpty()) {

Symbol sym = expression.removeFirst();

switch (sym.type) {

// sym is an OPERAND

case "<const>":

operands.push(Integer.parseInt(sym.lexeme));

break;

case "<iden>":

var = Variable.symbolTable.get(sym.lexeme);

operands.push(var.value);

break;

case "<literal>":

operands.push(sym.lexeme.substring(1, sym.lexeme.length()-1));

break;

// sym is an OPERATOR

default:

***Object*** **temp2** = operands.pop();

***Object*** **temp1** = operands.pop();

if (**temp1** ***instanceof Number*** && **temp2** ***instanceof Number***) {

***Number*** **operand1** = ***(Number)*** temp1;

***Number*** **operand2** = ***(Number)*** temp2;

if (**operand1** ***instanceof Integer*** && **operand2** ***instanceof Integer***) {

switch (sym.lexeme) {

case "+":

operands.push(**operand1**.***intValue()*** + **operand2**.***intValue()***);

break;

case "-":

operands.push(**operand1**.***intValue()*** - **operand2**.***intValue()***);

break;

}

} else {

switch (sym.lexeme) {

case "+":

operands.push(**operand1**.***doubleValue()*** + **operand2**.***doubleValue()***);

break;

case "-":

operands.push(**operand1**.***doubleValue()*** - **operand2**.***doubleValue()***);

break;

}

}

} else { // temp1 instanceof String || temp2 instanceof String

switch (sym.lexeme) {

case "+":

operands.push(**temp1**.***toString()*** + **temp2**.***toString()***);

break;

case "-":

operands.push(**temp1**.***toString()***.replaceFirst(**temp2**.***toString()***, ""));

break;

default:

operands.push(null);

break;

}

}

break;

}

}

return operands.pop();

}

}

// *may be used in assignment as...*

*Rule:* ***<assign> → <iden> <eq> <expr>***

*e.g.***num = num + 10 ;**

class Assign {

String iden;

**Expr expression;**

public Assign(Symbol lhs) {

iden = lhs.children.get(0).lexeme;

**Expr expression = new Expr(lhs.children.get(2));**

}

public void interpret() {

Variable x = Variable.symbolTable.get(iden);

**x.value** = **expression.interpret()**;

}

}