Project 4: java-python interpreter

0. 前言

0.1 python 解释器是什么?

Python 的解释器是一种可以执行 Python 代码的软件程序。Python 官方提供了多个解释器，包括 CPython、Jython、IronPython、PyPy 等。其中，CPython 是最常用的一个，也是官方默认的解释器。

0.2 python 解释器的执行过程（以 loop-and-eval 为例）

> "Loop-and-eval" 是一种常见的解释器实现模式，它的基本流程如下：

- 读取输入：从标准输入、文件或其他来源读取待解释的代码。
- 词法分析：将输入的代码分解成一个个 token，即词法单元，比如变量名、数字、运算符等。
- 语法分析：将 token 组成的序列转换成语法树，即把代码的结构化表示。
- 循环执行：进入一个循环，不断执行以下步骤，直到程序结束或遇到错误：
    - 从语法树中取出下一个表达式。
    - 对表达式进行求值，即执行表达式所表示的操作。
    - 如果表达式是一个控制流语句（如 if、while、for 等），则根据条件跳转到相应的代码块。
    - 如果表达式是一个函数调用，则进入函数执行过程。
    - 如果表达式是一个返回语句，则返回到调用该函数的位置。
- 输出结果：将程序的输出打印到标准输出、文件或其他目标。

在这个流程中，循环执行是解释器的核心部分，它负责执行代码并控制程序的流程。每次循环执行时，解释器从语法树中取出一个表达式，并对其进行求值。如果表达式是一个控制流语句，则根据条件跳转到相应的代码块；如果表达式是一个函数调用，则进入函数执行过程；如果表达式是一个返回语句，则返回到调用该函数的位置。通过这种方式，解释器可以逐步执行代码，并在必要时修改程序的状态，最终输出结果。

1. 作业说明

> 在本次作业中，我们将弱化其他过程，重点关注**循环执行**这个部分。

> 也就是说你不需要关心词法分析和语法分析具体是怎么进行的，你只需要理解分析的产物：由 token 组成的抽象语法树，并对其进行解释，输出执行结果。

> 本次作业待解释的代码来源是.py 文件，即我们将从.py 文件中读取待解释的 python 代码，我们会实现词法分析和语法分析部分和部分循环执行的例子，然后由你们对循环执行这一过程进行补充，最后在控制台输出结果。

==========

Getting Start

==========

1.1 代码结构

```
source                          //存放待解释的py文件
 --xxxtest.py

src.pers.xia.jpython
 --ast                          //存放抽象语法树节点
 --config
 --grammer                      //语法部分
 --interpreter                  //循环解释核心
   -- expression                //表达式类，其中的eval求值方法是关注重点
   -- statement                 //表达式语句类，其中的run方法是关注重点
   -- interpreter.java          //循环解释代码入口，其中parseAstNode方法是关注重点
   -- parser.java               //解析器类，其中parseExpression方法是关注重点
 --main                         //REPL代码
 --object                       //用java定义的python对象
 --parser                       //语法分析部分
 --tokenizer                    //词法分析部分
```

1.2 READ–EVAL–PRINT LOOP

> 我们提供了一种交互式的 python 解释器，你可以前往 main 函数中的 REPL.java 文件，并运行其主函数，就可以进入一个 REPL 的交互式终端进行 python 命令的调试。其核心部分是一个 READ–EVAL–PRINT 的循环结构，其循环体结构定义的伪代码结构如下：

```Java
while (true) {
     System.out.print(">>> ");
     String input = readInput();
     if (input.equals("exit")) {
          break;
     }
     try {
          evalLine(input);
     } catch (Exception e) {
          // Handle exceptions here
          e.printStackTrace();
     }
}
```

> 在这个循环体中，根据我们定义的 `readInput()` 函数，会循环地读取终端的输入数据，并设置了 `exit` 关键字用于退出循环,这部分会作为循环体的读入数据部分。

> eval–print 部分则全部集中在 evalLine 函数中，在这一部分中，我们仿照了实际 python 的 REPL 循环，定义了一个 while 循环去判断输入的语句是否完整，完整则执行命令，否则它会继续循环的读取终端的输入，直到语句完整。

```Java
PyReadMod mod = new PyReadMod(line);
while (!mod.isEnd()){
     //.....
     System.out.print("...");
     line = readInput();
     mod = new PyReadMod(line);
}
String formatStr = mod.format();
```

> 为了减轻在后续中词法分析的压力，使得通过终端交互输入命令与从文件读入时更为一致，我们构建了一个 PyReadMod 类，用于输入数据的规范化处理，这一部分中，你需要参与完成几个小的规范化步骤。

1.2.1 PyReadMod

> 在运行 REPL 循环后，你可以通过输入 `PYREAD` 关键字来进入该模式，在这里，你可以看到输入字符串规范化后的结果。PyReadMod 的循环体结构与上面的 REPL 循环非常类似,其中涉及到了两个关键的方法 `getIndentationLevel` 以及 `format`，这两个方法会在后文中进行介绍：

Java

```
        while (true) {
            System.out.print(">>>(Py-read) ");
            String input = readInput();
            if (input == null) {
                break;
            }
            if (input.equals("exit")) {
                pyReadMode = false;
                break;
            }
            try {
                PyReadMod mod = new PyReadMod(input);
                int num = mod.getIndentationLevel();
                String result = mod.format();
                System.out.println("(" + num + " " + result + ")");
            } catch (Exception e) {
                // Handle exceptions here
                e.printStackTrace();
            }
        }
```

> 这部分为 PyReadMod 的核心方法：

Java

```
private static class PyReadMod {

        private String line;

        public PyReadMod(String line) {
```

```
            this.line = line;
        }
        private String format(){
            StringBuilder sb = new StringBuilder();
            while (line.length()>0){
                Character c = peekChar();
                if(isNumber(c)&&sb.length()==0){ // number case
                    //...
                }else if(isComment(c)){          //comment case
                    //...
                }
                else if(isString(c)){           //string case
                    //...
                }else if(isDouble(c)){          //Double case
                    //...
                }else {
                    sb.append(readChar());
                }
            }
            return sb.toString();
        }
}
```

> PyReadMod 中持有了一个 String 类型的 line 变量用于存储每次从终端读入的命令，在本类中，我们提供了两个方法用于处理字符串，`peekChar()` 用于查看当前 `line` 的第一个字符的值，`readChar()` 会返回当前 `line` 的第一个字符，并将这个字符从 `line` 中移除。当 `line` 中没有新的字符之后，`peekChar()` 和 `readChar()` 方法都会返回一个空对象。

1.2.2 Task 1

> 在 format() 方法中，当在 `line` 中获取字符得到了 `#` 时，`isComment()` 方法会返回 `true`，并调用对应的 `ignoreComment()` 方法，现在的 ignoreComment 方法，不会对剩下的字符做任何的处理，请你实现当前的 `ignoreComment ()` 方法，使它能够吃掉所有剩下评论词，而不将其加入到需要执行的命令中。

Java

```
>>>(Py-read) print(12) #123
(0 print(12) #123)
// error answer
```

```
>>>(Py-read) print(12) #123
(0 print(12))
//correct answer
```

1.2.3 Task2

> 你可能已经发现了，在 `Py-read` 模式时，终端打印的内容中附加上了一个 0 的记号，这是因为在 python 中使用了缩进的方式来进行对各个代码块内容的分割的，在本次任务中，你需要去实现 mod 中的 `getIndentationLevel()` 方法，这个方法会循环的读取 `line` 字符串的首个字符，并去判断它是否为空格，最终得到当前代码的缩进空格数。

注意： 所有的 `Tab` 都会转换为 4 个空格。

```Java
>>>(Py-read)   print(123)
(4 print(123))
>>>(Py-read)  x=3
(1 x=3)
```

1.2.4 Task3

> 接下来我们需要对 String 类型进行处理，当在 `line` 中获取字符得到了 `'` 或者是 `"` 时， `isString()` 方法会返回 `true` ，你需要实现 `getFormatString(character ch)` 方法来获取一个完整的字符串序列，方法的入参 `ch` 是字符单引号或双引号，你需要根据单引号与双引号的情况，来循环的读取字符，直到遇到下一个对应的引号。现在的实现在遇到引号时会直接抛出异常，提醒你方法还未实现。

注意：当然，字符串中还存在着转义字符的类型，本次实验中也支持了转义字符，由于某些转义字符会在词法分析时产生冲突，这里使用的做法是直接存储，如 `"\n"` 在实际存储时会被存为 `\` + `n` ，在最后的解析部分，再做转义的变换，转义字符的内容将不会在测试部分中进行测试。

```Java
>>>(Py-read) print("123")
(0 print("123"))
>>>(Py-read) print('ab"c"')
(0 print('ab"c"'))
```

另外，我们也提供了使用**文件读入**的方式来进行 python 代码解释的功能，下面我们直接就一个例子来进行展开论述。

1.3 python Interpreter 示例

> 我们通过一个简单的例子来看下代码的执行流程，首先创建一个待解释的 python 文件

Python

```python
# 输入文件
n = 1
print(n)
```

assigntest.py

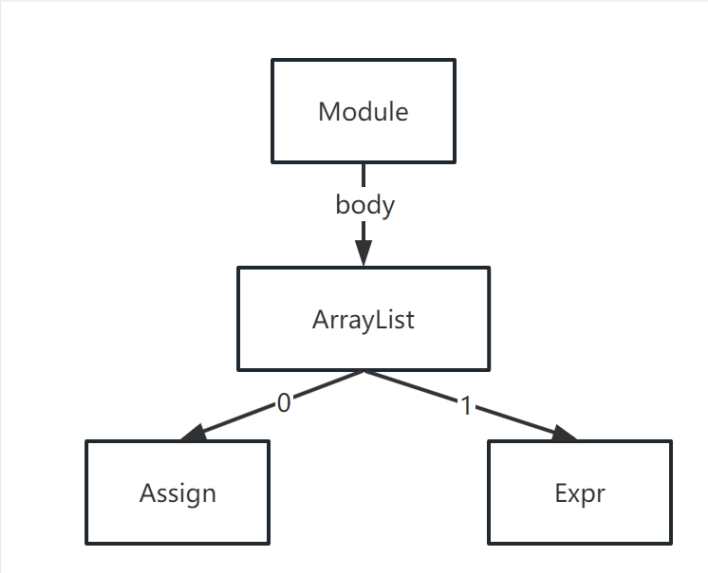
> 然后，我们来到 `interpreter.java` 文件，找到 `main` 函数，这是整个解释器代码的入口。

Java

```java
    public static void main(String[] args) {
        //读取文件
        String fileName = "./source/assigntest.py";
        File file = new File(fileName);
        try
        {
            // tokenizer符号化
            Node node = ParseToken.parseFile(file, GramInit.grammar, 1
            Ast ast = new Ast();
            // 生成抽象语法树
            Module mod = (Module) ast.fromNode(node);
            // 解释执行
            Interpreter interpreter = new Interpreter(mod.getBody());
            interpreter.runProgram();
        }
        catch (PyExceptions e)
        {...}
    }
```

interpreter.java

我们只需要关注解释执行部分，不妨在 13 行打个断点，看看 `mod.getBody()` 获取到的 AST 抽象语法树是什么结构。如下图所示，这个 `body` 是一个数组，两个节点分别对应两条语句：`Assign` 和 `Expr`



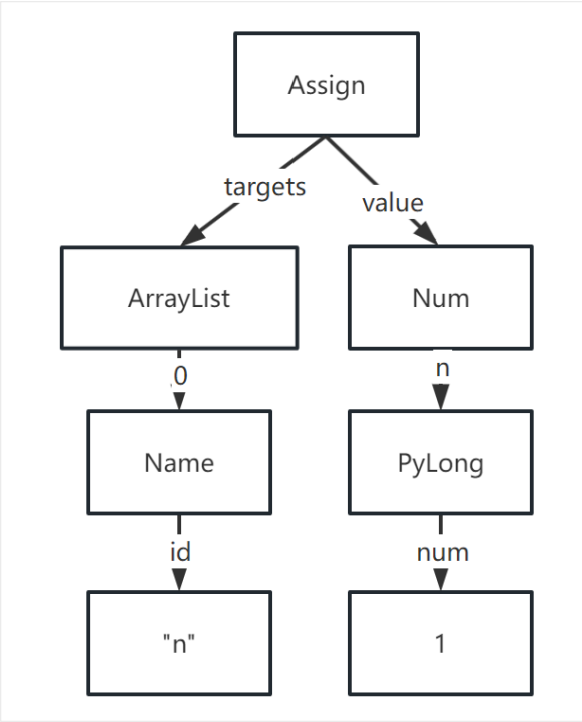
在 `interpreter` 的构造函数中，我们遍历上一步得到的 `body`，使用 `parseAstNode` 对节点进行处理，并将得到的可执行语句储存到解释器中

Java

```java
public class Interpreter {
    private final List<Statement> statements = new ArrayList<>();
    public static ProgramState programState = new ProgramState();
    ...
    public Interpreter(List<stmtType> nodes){
        for (stmtType node : nodes) {
            Statement statement = parseAstNode(node);
            statements.add(statement);
        }
    }
    ...
}
```

interpreter.java

在 `parseAstNode` 方法中，我们会判断节点的类型，然后提取节点的有效信息，返回一个可执行的语句。比如 `Assign` 节点，我们可以在 `ast` 包中找到 `Assign.java` 文件查看定义，就能知道它有一个被赋值对象 `targets`，有一个赋值对象 `value`。



（同样，你也可以在下面的代码第 4 行打一个断点，就能知道这个 node 的结构，**这个操作在后续写代码的过程中非常有用，能帮助我们快速直观感知一个 node 的结构**），因为赋值语句在执行时会对右值进行计算，所以我们需要用 `Assign.value` 创建一个 `expression`，在 `AssignStatement` 初始化的时候传入。

Java

```
1   private Statement parseAstNode(stmtType node){
2       ...
3       if(node instanceof Assign){
4           String variableName;
5           exprType target = ((Assign) node).targets.get(0);
6           variableName = target.getId();
7           exprType value = ((Assign) node).value;
8           Expression expression = parser.parseExpression(value);
9           return new AssignStatement(variableName,expression);
10      }
11      ...
12  }
```

Interpreter.java

> `expression` 的意义是，对不同的 `expression` 定义不同表达式的求值方法，当调用 `expression.eval(programeState)`，就可以获取表达式的值。

- 比如 `ConstantExpression` 会返回创建时传入的常量。
- 而 `VariableExpression` 会根据变量名到 `programState` 中查找当前值。
- `BinOpExpression`（二元运算表达式）会首先调用 `leftExpression.eval()` 和 `rightExpression.eval()`，然后再运算。

> 在本例中，`value` 是一个 `Num` 类型的常量，因此 `parser.parseExpression(value)` 会返回一个 `ConstantExpression`。

Java

```java
public Expression parseExpression(exprType expression){
        ....
        if(expression instanceof Num){
            Num n = (Num) expression;
            return new ConstantExpression(n.n);
        }
        ....
}
```

Paser.java

> `ConstantExpression` 的定义如下

Java

```java
public class ConstantExpression extends Expression{

    public ConstantExpression(PyObject n) {
        super(n); // super执行的就是this.val = n
    }
    @Override
    public PyObject eval(ProgramState programState) {
        return this.val;
    }
}
```

ConstantExpression.java

> 当我们成功将每一个 AST 节点转化为 `statement` 之后，还记得 `main` 函数中的 `interpreter.runProgram()` 吗，他会遍历每一条 `statement` 执行

Java

```
public class Interpreter {
    ...
    public void runProgram() {
        for (Statement statement: statements) {
            statement.run(programState);
        }
    }
    ...
}
```

Interpreter.java()

> 我们来看看刚刚创建的 `AssignStatement` 会做些什么，可以看到，他首先对右值进行计算，然后对变量进行赋值。

Java

```
public class AssignStatement implements Statement {
    private String variableName;
    private Expression expression;

    public AssignStatement(String variableName, Expression expression)
        this.variableName = variableName;
        this.expression = expression;
    }

    @Override
    public void run(ProgramState programState) {
        PyObject expressionValue = expression.eval(programState);
        programState.setVariable(variableName, expressionValue);
    }
}
```

> 我们再来看看 `expreStatement` 会做些什么？可以看到，`expreStatment` 简单的对表达式进行求值，然后输出。

Java

```
public class ExprStatement implements Statement {
    private Expression expression;

    public ExprStatement(Expression expression) {
        this.expression = expression;
    }

```

```
    @Override
    public void run(ProgramState programState) {
        if(expression.eval(programState)!=null){
            System.out.println(expression.eval(programState));
        }
    }
}
```

> 最终，程序执行完成后，我们会在控制台看到如下输出

纯文本

```
1

Process finished with exit code 0
```

这就是一个简单的执行流程了，纸上得来终觉浅，同学们可以把代码跑起来，自己 debug 跟一遍流程。在程序执行中，可以重点关注 `parseAstNode`，`paserExpression`，`eval`，`run` 这四个方法，理解 `ast` 包中重要的类型，并且对 Object 中常用 `PyObject`（比如 `PyLong`，`PyFloat`，`PyBoolean`，`PyUnicode(就是str)`）进行学习。其他难以理解的细节，可以在后续编码过程逐渐深入理解。

1.3 Task 1

在阅读刚刚的例子时，你可能发现一个问题，这里的 `Assign.targets` 为什么是一个 `ArrayList`，且为什么只取第一个进行赋值呢?

Java

```
  private Statement parseAstNode(stmtType node){
      ...
      if(node instanceof Assign){
          String variableName;
          // 为什么只取第一个进行赋值呢？
          // Todo: 实现能够连续赋值，比如a=b=1
          exprType target = ((Assign) node).targets.get(0);
          variableName = target.getId();
          exprType value = ((Assign) node).value;
          Expression expression = parser.parseExpression(value);
          return new AssignStatement(variableName,expression);
      }
```

```
    ...
}
```

Interpreter.java

这是因为，类似于 `a=b=1` 这样的连续赋值语句，会有超过一个 `target` (a 和 b)，所以在生成抽象语法树时，`Assign.targets` 会是一个 `ArrayList`；

至于第二个问题，正是我们需要完成的第一个 `Task`，按照目前的实现，对于 `a=b=1` 的赋值语句，程序会漏掉对 `b` 的赋值，从而出现错误，因此，你需要修改代码，以实现连续赋值。

> tips：既然要赋多个值，那么将传入的 `variableName` 类型改为一个数组然后进一步修改 `AssignStatment` 即可

1.4 Task 2

下面是一段二元运算 `BinOpExpression` 的代码，目前只实现了 `Add(+)`，`Sub(-)`，`Mult(*)` 三种运算，你需要完成 `Mod(%)`，`Div(/)`，`FloorDiv(//)` 三种运算。

Java

```java
public class BinOpExpression extends OpExpression {
    OperatorType operatorType;
    public BinOpExpression(Expression lhs, Expression rhs, OperatorType
    {
        super(lhs, rhs);
        this.operatorType = operatorType;
    }

    @Override
    public PyObject eval(ProgramState programState) {
        PyObject lhs = this.lhs.eval(programState);
        PyObject rhs = this.rhs.eval(programState);
        switch (operatorType){
        // TODO: 完成Mod(%)，Div(/)，FloorDiv(//)三种运算
            case Add:
                return lhs.add(rhs);
            case Sub:
                return lhs.sub(rhs);
            case Mult:
```

```
                return lhs.mul(rhs);
            default:
                return new PyString("暂未实现");

        }
    }
}
```

BinOpExpression.java

> ps:除了修改 `BinOpExpression`，在 `PyObject` 内部也需要实现对应的方法。具体可以参考 `Add`， `Sub` 和 `Mult` 的实现，当然也鼓励大家探索其他方式！

> 同时，需要在父类 `PyObject` 中定义兜底方法，当类型不支持时，会调用 `super` 的对应方法抛出错误，比如下面 `mul` 的例子：

Java

```
public abstract class PyObject
    public PyObject mul(PyObject p){
        throw new PyExceptions(PyExceptions.ErrorType.TYPE_ERROR, "TypeE
    }
}
```

Java

```
// Class PyUnicode
public class PyUnicode extends PyObject{
    @Override
    public PyObject mul(PyObject p) {
        // str只能和整数相乘
        if(p instanceof PyLong){
            return new PyUnicode(String.join("", Collections.nCopies((i
        }
        else{
            super.mul(p);
            return new PyNone();
        }
    }
}

// Class PyFloat
public class PyFloat extends PyObject{
    @Override
```

```
19      public PyObject mul(PyObject p) {
20          if (p instanceof PyLong){
21              return new PyFloat(this.num * ((PyLong) p).asLong());
22          }
23          if (p instanceof PyFloat){
24              return new PyFloat(this.num * ((PyFloat) p).asFloat());
25          }
26          if(p instanceof PyBoolean){
27              return new PyFloat(this.num * ((PyBoolean) p).asInt());
28          }
29          else{
30              super.add(p);
31              return new PyNone();
32          }
33      }
34  }
35
36  // Class PyLong
37  public class PyLong extends PyObject{
38      @Override
39      public PyObject mul(PyObject p) {
40          if (p instanceof PyLong){
41              return new PyLong(this.num * ((PyLong) p).asLong());
42          }
43          if (p instanceof PyFloat){
44              return new PyFloat(this.num * ((PyFloat) p).asFloat());
45          }
46          if(p instanceof PyBoolean){
47              return new PyLong(this.num * ((PyBoolean) p).asInt());
48          }
49          if(p instanceof PyUnicode){
50              return new PyUnicode(String.join("", Collections.nCopies((i
51          }else{
52              super.mul(p);
53              return new PyNone();
54          }
55      }
56  }
```

1.5 Task 3

为了简化 for 语句，目前的程序只实现了 `for x in range(n)` 或者 `for x in range(a,b)` 这两种形式，请你补充实现 range 有 3 个参数，即 `for x in range(start,end,step)` 的情况，以实现不同步长。

Java

```java
    private Statement parseAstNode(stmtType node){
        if (node instanceof For){
            For forNode = (For) node;
            if(!(forNode.target instanceof Name) || !(forNode.iter inst
                return new EmpytStatement();
            }
            String variableName = ((Name) forNode.target).getId();
            Call range = (Call) forNode.iter;
            if(!(range.func instanceof Name)  || !((Name) range.func).g
                return new EmpytStatement();
            }
            Expression start = range.args.size() > 1 ? parser.parseExpr
            Expression end = range.args.size() > 1 ? parser.parseExpres

            //Todo：从range中提取第3个参数

            List<Statement> body = new ArrayList<>();
            List<Statement> elseBody = new ArrayList<>();
            for(stmtType statement: forNode.body ){
                body.add(parseAstNode(statement));
            }
            if(forNode.orelse != null) {
                for(stmtType stmt: forNode.orelse){
                    elseBody.add(this.parseAstNode(stmt));
                }
            }
            return new ForStatement(variableName,start,end,body);
        }
    }
```

Java

```java
 // Todo: 修改BlockStatement实现不同步长
public class ForStatement extends BlockStatement {
    private String loopVariable;
    private Expression rangeStart;
    private Expression rangeEnd;

```

```
    public ForStatement(String loopVariable, Expression rangeStart, Exp
        super(null, body);
        this.loopVariable = loopVariable;
        this.rangeStart = rangeStart;
        this.rangeEnd = rangeEnd;
    }

    @Override
    public void run(ProgramState programState) {
        programState.setVariable(loopVariable, rangeStart.eval(programS
        for(long i = ((PyLong)programState.getVariable(loopVariable)).a
            boolean ifBreak = bodyBlock(programState);
            if(ifBreak){
                return;
            }
            programState.setVariable(loopVariable, new PyLong(i+1));
        }
        elseBlock(programState);
    }
}
```

ForStatement.java

1.6 Task 4

我们定义了 `BlockStatment` 用来实现 `If块，for块，while块`，目前我们已经实现了 `If` 和 `for`，请你根据 `While` 类型的结构完成 `parseAstNode` 的 `while` 部分，并且根据 `while` 的特点利用已经定义好的 `bodyBlock` 和 `elseBlock` 实现 `BlockStatement` 的 `whileLoop` 方法。

Java

```
    private Statement parseAstNode(stmtType node){
        if(node instanceof While){
              While whileNode = (While) node;
              //Todo 实现对While的解析
              return new WhileStatement(test,body,elseBody);
          }
    }
```

Interpreter.java

Java

```
public abstract class BlockStatement implements Statement{
```

```
    private Expression expression;
    private List<Statement> body;
    private List<Statement> elseBody;

    public void whileLoop(ProgramState programState) {
        // TODO: 完成whileLoop实现while循环
    }
}
```

BlockStatement.java

> tips：可以参考 `if`，`for` 的实现，最终实现效果如下

Python

```
n = 0
while n<10:
    n=n+1
    if n == 5:
        break
    if n == 2:
        continue
    print(n)

# 控制台输出
1
3
4
```

1.7 Task 5

在语法解析过程中，有一种表达式类型为 `BoolOp`，即逻辑运算。你需要完成对 `BoolOp` 的结构进行解析，然后返回一个 `BoolOpExpression`，并补充 `BoolOpExpression` 的 `eval` 方法，返回逻辑运算结果

Java

```
public class Parser {
    public Expression parseExpression(exprType expression){
        if(expression instanceof BoolOp){
            // TODO: 完成对BoolOp的解析
            return new BoolOpExpression(expressions,boolOp.op);
        }
    }
}
```

Parser.java

```java
public class BoolOpExpression extends Expression{
    List<Expression> values; // 参与逻辑运算的表达式
    boolopType type; // 逻辑运算类型 boolopType.And/boolopType.Or
    public BoolOpExpression(List<Expression> values, boolopType type){
        this.values = values;
        this.type = type;
    }
    @Override
    public PyObject eval(ProgramState programState) {
        boolean res;
        // TODO 补充代码返回逻辑运算结果
        return new PyBoolean(res);
    }
}
```

BoolOpExpression.java

> tips：在 `eval` 中，你需要对 `values` 中的表达式分别调用 `eval()` 方法
>
> 可能会用到我们已经写好的 `PyObject` 的 `asBoolean()` 方法。