**编译项目报告**

卫艺璇 15307130433

王炜越 15307130349

1. **ANTLR 4与Yacc/Bison工具的对比**

* ANTLR生成LL(k)解析器，而Yacc和Bison都生成LALR解析器。LL(k)语法相对于LALR更加强大，也没有shift-reduce，reduce-reduce类似的语法冲突错误。
* ANTLR生成代码后可读性很高，因其全部封装在对应的class中。在语法的parser上，ANTLR比较清晰的switch/case来匹配token，类似于手动书写一个DFA，而在Yacc中使用的是parser table。并且对于switch的效率问题，因为编译器对于switch是有优化的，所以效率上没有很大影响。
* ANTLR提供了对于tree construction、tree walking和translation的支持，这对于我们话抽象语法树有很大的帮助。
* ANTLR的错误处理机制很灵活，其使用exception-driver实现，exception的最小粒度能得到具体的token，方便我们在语法解析时控制系统的错误处理。
* ANTLR对于不同语言有很好的支持，对于一个rule，我们可以很方便地加入一段代码，同时可以很方便地在规则之外加入类的成员变量、类的成员函数以和全局的变量和函数等。
* ANTLR不仅功能更强、容易扩展和开源，而且ANTLR生成的代码和使用递归下降方法（手工生成分析器的主要方法）生成的代码很相似，易于阅读理解。而基于LR分析法的Yacc分析器生成工具生成的程序就比较晦涩。此外我们可以在文法描述中插入特定的语义动作，告诉ANTLR怎样去创建抽象语法树和怎样输出（对我们PJ的完成很有帮助）。

1. **项目流程**

开发环境：macOS Mojave, jdk-1.8.0, ANTLR-4.7.2, IntelliJ IDEA

1. 首先书写一个MiniJava.g4文件，该文件中定义了对于mini java语言[[1]](#footnote-1)的ANTLR格式的语法Parser和词法Lexer。
2. 在命令行中执行java -jar /下载地址/antlr-4.7.2-complete.jar命令调用ANTLR库来编译MiniJava.g4文件。成功执行之后，可以得到如下几份文件。安装ANTLR和设置环境变量的过程详见官网教程[[2]](#footnote-2)。

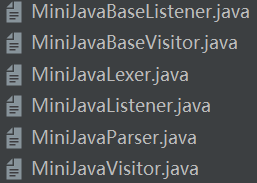


图1: ANTLR生成的java文件

1. 根据自己需求修改这些文件，并再新建一个MiniJavaAnalyze.java文件和一个ErrorListener.java文件，分别用来结合这些文件生成抽象语法树以及进行错误处理，其中前者定义了整个程序的主函数。
2. 然后利用javac MiniJava\*.java ErrorListener.java编译对应.java文件生成相应.class文件，最后使用java MiniJavaAnalyze运行编译好的MiniJavaAnalyze.class文件。
3. 测试：输入展示模式的选择（1代表语法树中展示全部代码文本，0代表仅展示叶子结点文本），以及MiniJava格式的源代码，Command+D结束输入，该程序会运行生成抽象语法树的图，并以log形式在命令行中进行语法错误的显示。
4. **源代码分析**

3.1 **源代码结构**

源代码文件夹src/MiniJava中包含8个文件：

* ErrorListener.java：我们用来错误处理的代码。
* MiniJavaAnalyze.java：核心代码，定义了main函数，用来生成抽象语法树。
* MiniJavaLexer.java：ANTLR中的句法Lexer处理文件
* MiniJavaParser.java：ANTLR中的Parser处理文件
* MiniJavaListener.java：ANTLR自动生成的监听函数。
* MiniJavaVisitor.java：ANTLR自动生成的访问函数。
* MiniJavaBaseListener.java：ANTLR自动生成的MiniJavaListener的监听函数实例。
* MiniJavaBaseVisitor.java：ANTLR自动生成的MiniJavaVisitor的访问函数实例。

**3.2 核心代码工作原理**

1. 在核心代码MiniJavaAnalyze.java中，我们首先使用ANTLR中的句法分析器lexer和语法分析器parser（如下图2）处理输入的mini java代码（如下图3），得到一个初始的非常啰嗦的语法树Parser Tree如下图4：

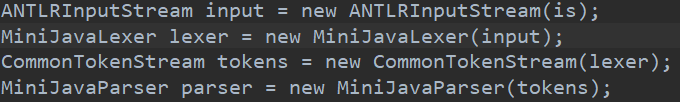


图2: 生成ANTLR自带句法语法分析器

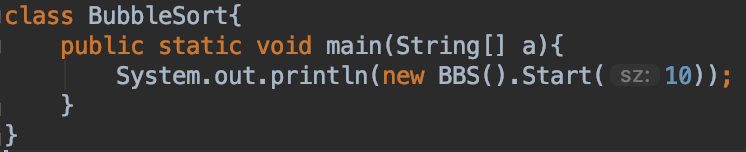


图3: 示范mini java代码

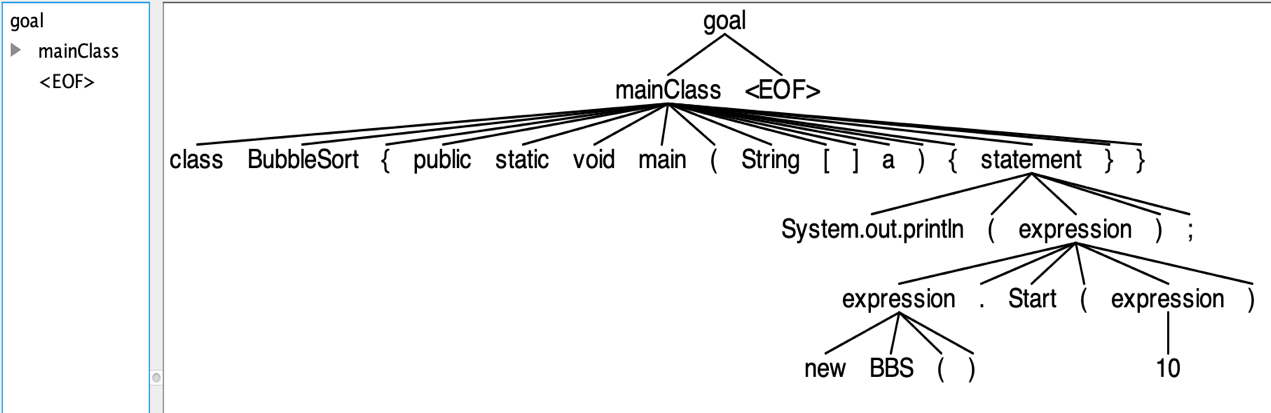


图4: ANTLR自带的gui界面和语法树

1. 错误处理：

新建自定义的错误处理文件：ErrorListener.java，继承了ANTLR自带的DiagnosticErrorListener类，其重写了syntaxError方法，加入了语义分析错误栈的打印和对错误位置的下划线标注这两种功能（效果如下图6），将原本语法分析器的错误监听函数替换为这个新类（如下图5）；此外，通过设置parser.getInterpreter().setPredictionMode()函数将文法中所有有二义性的地方都显示出来（不过因为我们定义的语法没有二义性，所以这个功能无法展示）。

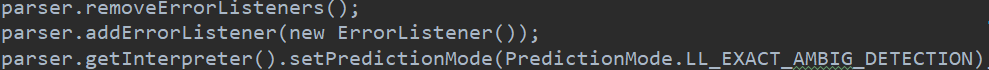


图5: 自定义的错误处理

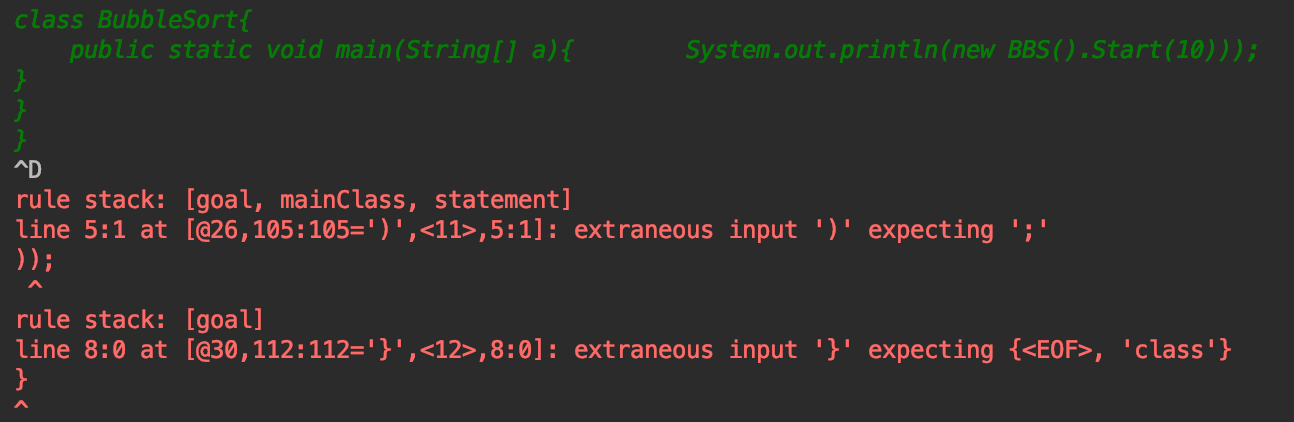


图6: 下划线标注错误位置和语法错误栈

而错误修复则利用了ANTLR自带的错误修复机制：在匹配代码时，遇到无法匹配的情况（多了或者少了一个符号）会尝试丢弃或者自动添加该符号，并继续匹配，如果同时遇到两个符号的错误，则会对这一语法规则放弃匹配，再从下一同级或更高的语法规则匹配（如图7中，错误的10和ijj之后，会退回更高层的.Start()语法继续匹配）。

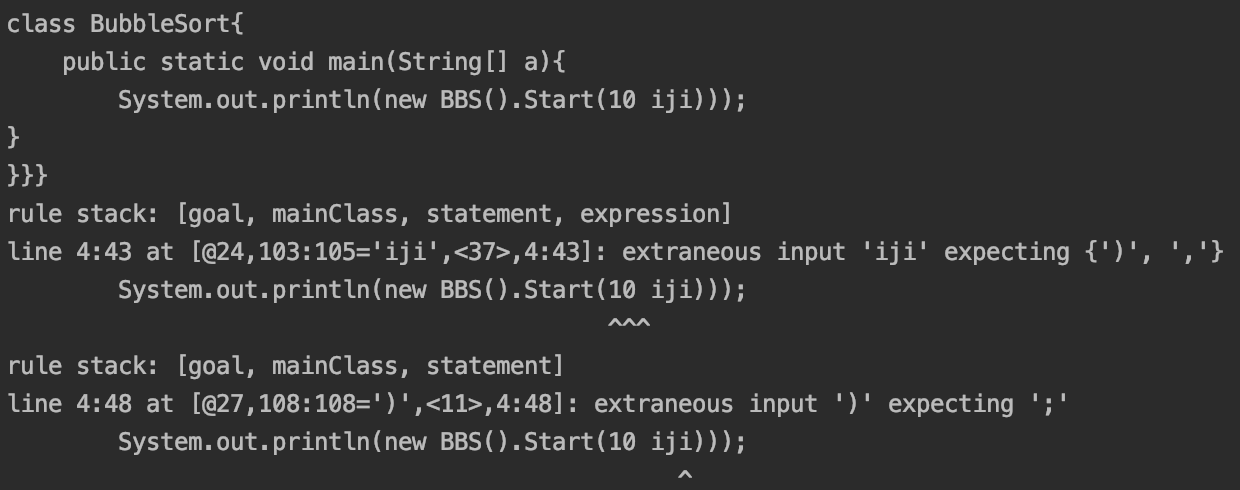


图7: 错误修复机制

1. 接下来是生成抽象语法树AST：

A. 先利用parser.goal()得到一个ANTLR 语法树的根结点（类型为ParserRuleContext），然后调用自定义的generateAST函数画出相应抽象语法树（如下图8）。



图8: 生成AST的部分代码

B. 画抽象语法树的主要思想是：

对ANTLR生成的语法树进行递归遍历。

语法树有两种结点：叶子结点TerminalNodeImpl和非叶子结点RuleContext。如果遍历到非叶子结点，则继续递归调用generateAST向下遍历，在下一层调用中根据是否是必要的结点（ignored为假则是必要的），建立相应的结点（如下图9）；如果遍历到叶子结点，则利用结点的Token Type（Token.getType()，返回的整数代表定义在MiniJavaLexer.java中的ruleNames数组）筛选出没被忽略的自定义的符号（如g4文件我们定义了Identifier和INT），建立相应的结点（如下图10）。

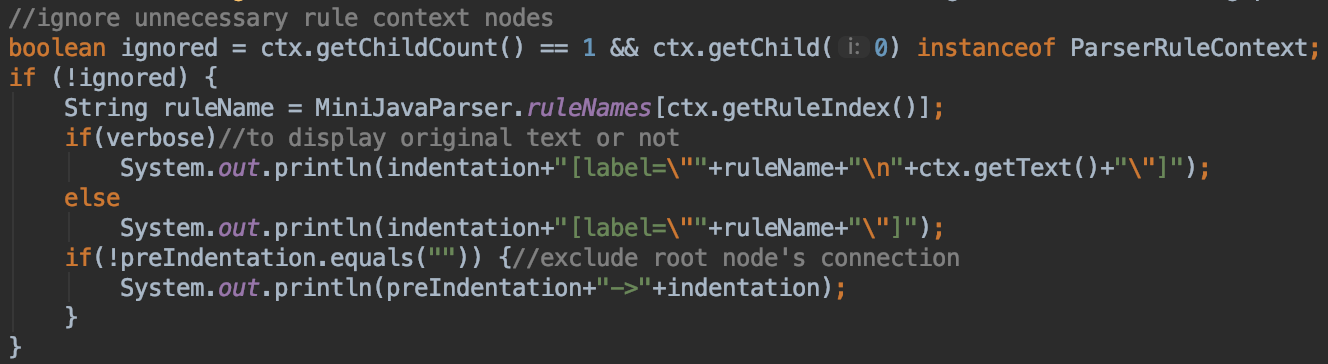


图9: 非叶子结点的打印过程

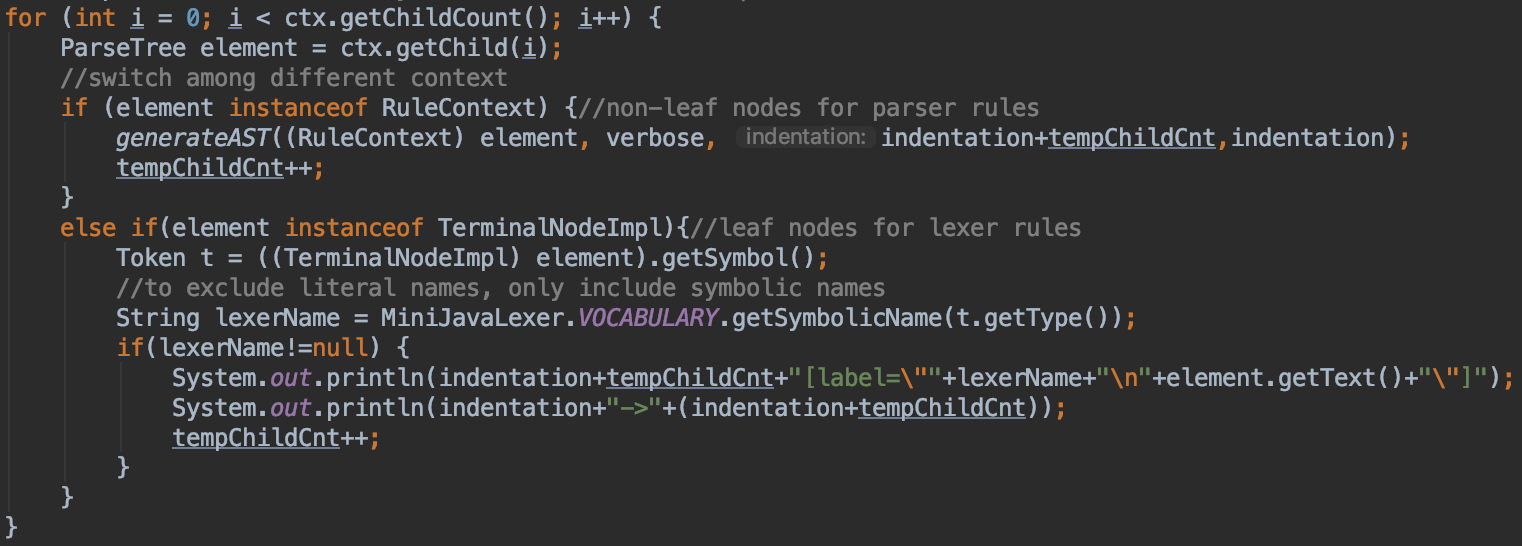


图10: 对一个非叶子结点所有子结点按类型讨论

对非叶子结点，还有由用户定义的verbose布尔值（如图11），规定是否需要在抽象语法树中展示非叶子结点的全部代码文本（verbose为true和为false的效果分别如图12，图13所示）。

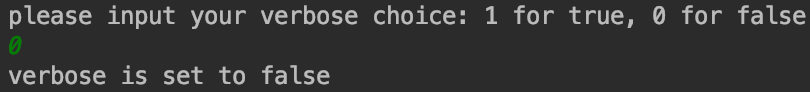


图11: 请求用户输入verbose布尔值

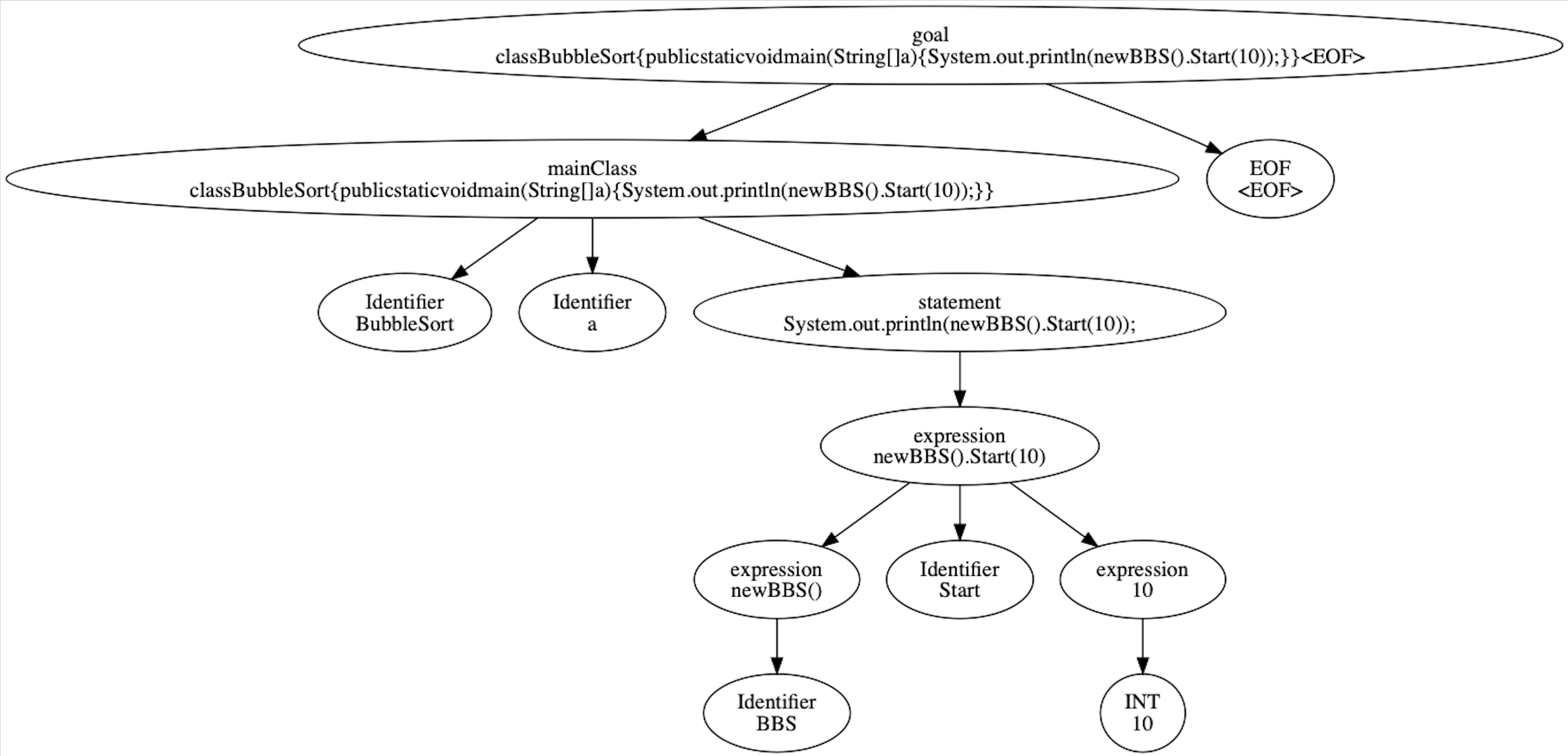


图12: verbose置为true的语法树效果图

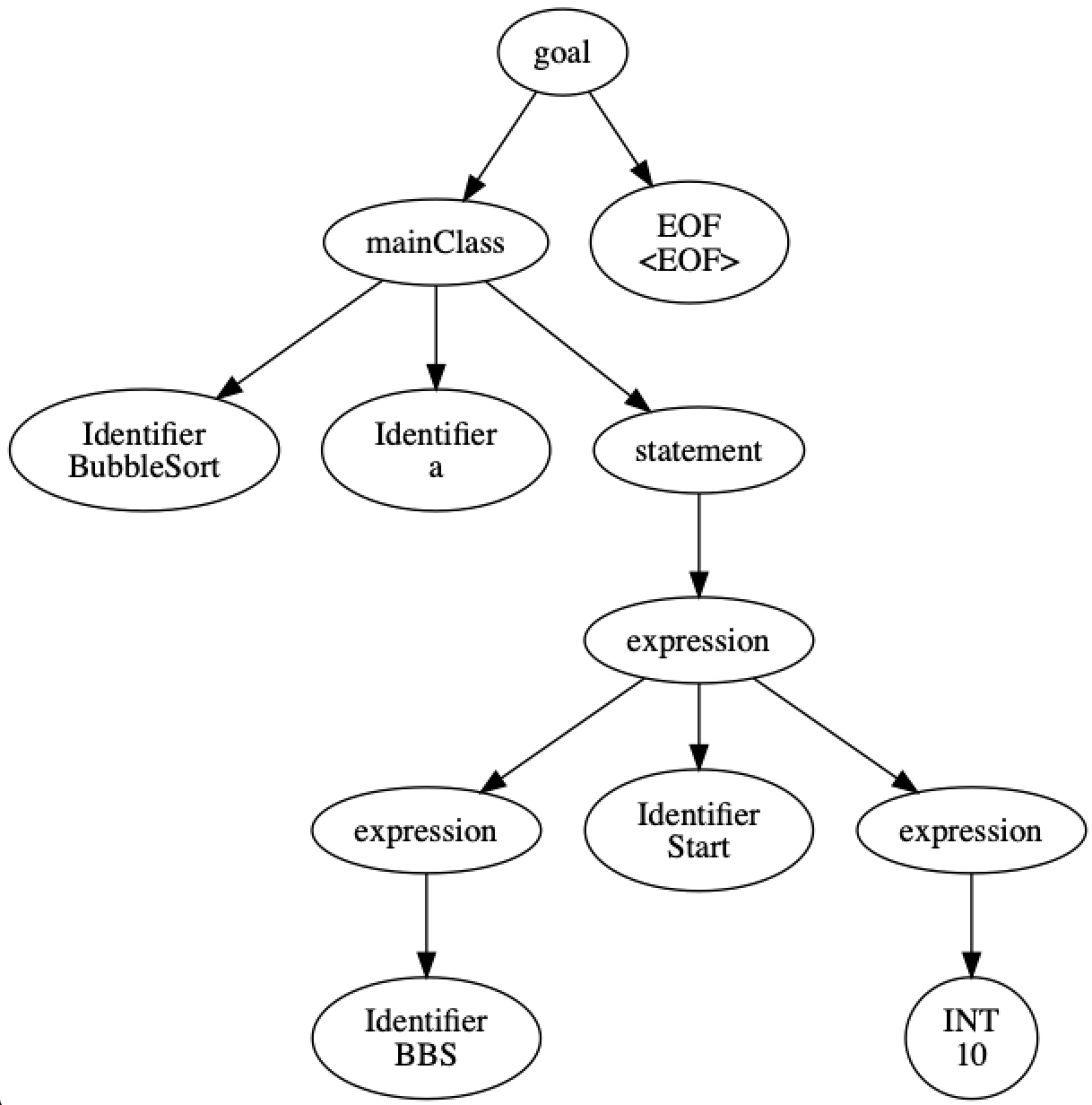


图13: verbose置为false的语法树效果图

而画图使用的是graphviz[[3]](#footnote-3)第三方软件接口，也就是按照需求生成dot语言[[4]](#footnote-4)书写的test.dot文件，再执行命令行的graphviz画图命令生成语法树图graph.png，并打开图片。代码如下图14所示。graphviz相关软件和生成的test.dot都在src/draw文件夹中。

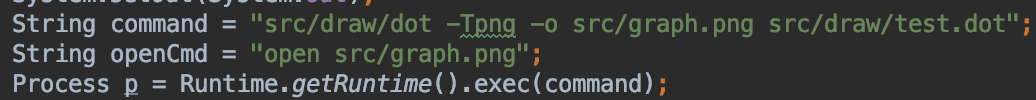


图14: 运用graphviz画图的代码

由于dot语言对于每个结点都有独特的标识来进行连接和赋上样式，我们采用了一种递增式的字符串编码方式：每次generateAST调用都传递当前要递归的非叶子结点ctx，前述的verbose变量，当前字符串标志indentation和其父母结点的字符串标志preIndentation，如图15所示。对当前非叶子结点遍历的时候（如图10所示），定义了递增的int变量tempChildCnt，给每个孩子会赋予当前结点下独特的标识，即每个孩子的indentation会变为其父母的indentation+tempChildCnt。比如图13中的Identifier BubbleSort叶子结点的标识就是000，Identifier Start标识则是00201，而根结点遍历时，preIndentation置为空，Indentation则为0.



图15: generateAST的变量定义

**3.3 工作中遇到的问题及解决**

1) ANTLR4的安装和配置

在尝试进行ANTLR官网教程的时候，总是会遇到java classpath没有正确设置的情况，也即每次运行必须使用-cp指定ANTLR jar包的位置，在mac中设置的系统环境变量总是失败。后来经过摸索，尝试了从重启终端到重装jdk到重启电脑到重装ANTLR包到系统文件夹的各种步骤之后，终于总结出了正确的设置：A. 下载ANTLR jar包到系统文件夹而非用户文件夹，下载版本不小于1.8的jdk；B. 在~/.bash\_profile中设置CLASSPATH，一定要包含当前目录”.”，以及antlr4和grun两个命令的alias方便后续操作；C. 命令行中执行source ~/.bash\_profile引入新设置的环境变量。

2) 匹配语法的顺序问题

在测试项目程序的时候，发现形如“2+1\*3”的匹配是有问题的，“2”和“1”被匹配成了二元运算的两端，而“2+1”和“3”被匹配成了更高一层的二元运算的两端。仔细审查之后，推断是因为ANTLR4的语法规则是默认从上到下匹配的，而在此之前，我们只是将一个语法规则的所有子项随意排列，乘法在加法前面，因而有这样的错误。我们因此调整了所有的语法规则子项的排列顺序，尤其是对expression中各计算符的顺序调整（调整后结果如下图16所示）。



图16: expression的匹配规则的正确顺序

1. **额外功能的说明与项目感想**

项目感想：

通过本次项目，我们更加理解了编译器的工作原理，将ANTLR生成的语法分析器Parser和词法分析器Lexer构造出的语法树parser tree进一步抽象为机器更好理解的抽象语法树。项目的前期花了很多时间在ANTLR的语法学习和mac上java开发环境上面，因为我们对这两者都不够熟悉，后来改用了IntelliJ IDEA也是因为在命令行中对java编译调试太过繁琐。不过好的工具确实事半功倍，相比于其他选择从零搭建的同学，直接使用包装好了的基于左递归的ANTLR包是还是十分快捷方便的。

项目的主要任务是构建抽象语法树，相比于代码实现，搞明白什么是抽象语法树才是真正的难点。。。

错误检查和修复主要参考了“The Definitive ANTLR4 Reference 2013”这本书。在真实的编译器中，还需要考虑提示不匹配的语法规则之外更隐秘的错误，比如数组越界、内存访问冲突、变量未声明/类型不匹配，这些需要真正建立变量符号表的树并模拟运行才能发现的错误。因为时间和水平的限制，这些问题就只能留待以后再探索了，网络上也有很多相关的参考书籍和资料。

总之，从这个尚不成熟的Mini Java编译器前端项目中，我们更好的理解了编译课程中提到的许多概念，比如词法、句法、语法分析还有语法匹配，也积累了更多的开发经验，相信这会对我们以后的学习、工作有深远的帮助和意义。

1. <http://www.cambridge.org/us/features/052182060X/grammar.html> [↑](#footnote-ref-1)
2. <https://www.antlr.org/> [↑](#footnote-ref-2)
3. <http://www.graphviz.org> [↑](#footnote-ref-3)
4. <https://graphviz.gitlab.io/_pages/doc/info/lang.html> [↑](#footnote-ref-4)