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）。

2. 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]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]。

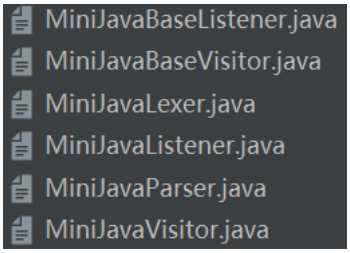

图 1: ANTLR 生成的 java 文件

3) 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。

[1] http://www.cambridge.org/us/features/052182060X/grammar.html
[2] https://www.antlr.org/

4) 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。
5) 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。

3. 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：

```
ANTLRInputStream input = new ANTLRInputStream(is);
MiniJavaLexer lexer = new MiniJavaLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
MiniJavaParser parser = new MiniJavaParser(tokens);
```

图 2: 生成 ANTLR 自带句法语法分析器

```
class BubbleSort{
    public static void main(String[] a){
        System.out.println(new BBS().Start( sz: 10));
    }
}
```

图 3: 示范 mini java 代码

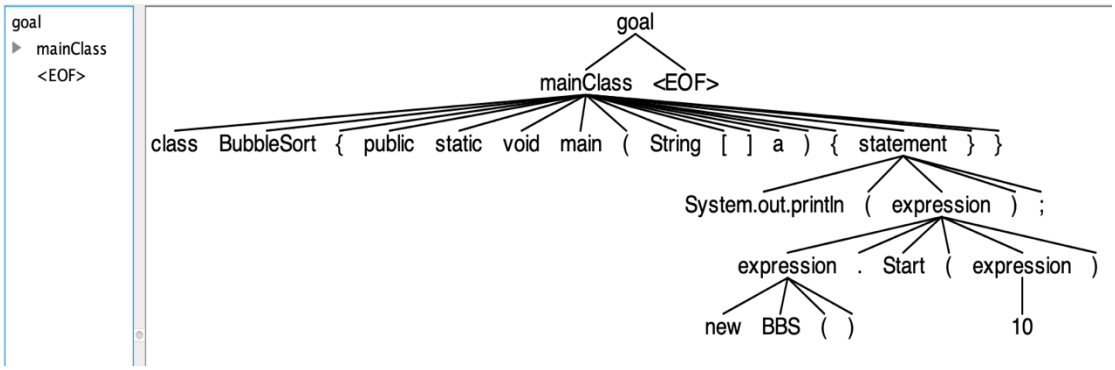

图 4: ANTLR 自带的 gui 界面和语法树

2) 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）。

```
parser.removeErrorListeners();
parser.addErrorListener(new ErrorListener());
parser.getInterpreter().setPredictionMode(PredictionMode.LL_EXACT_AMBIG_DETECTION)
```

图 5: 自定义的错误处理

```
class BubbleSort{
    public static void main(String[] a){        System.out.println(new BBS().Start(10)));
}
}
}
^D
rule stack: [goal, mainClass, statement]
line 5:1 at [@26,105:105=')',<11>,5:1]: extraneous input ')' expecting ';'
));
 ^
rule stack: [goal]
line 8:0 at [@30,112:112='}',<12>,8:0]: extraneous input '}' expecting {<EOF>, 'class'}
}
^
```

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）。

```
class BubbleSort{
    public static void main(String[] a){
        System.out.println(new BBS().Start(10 iji)));
}
}}}
rule stack: [goal, mainClass, statement, expression]
line 4:43 at [@24,103:105='iji',<37>,4:43]: extraneous input 'iji' expecting {')', ','}
        System.out.println(new BBS().Start(10 iji)));
                                           ^^^
rule stack: [goal, mainClass, statement]
line 4:48 at [@27,108:108=')',<11>,4:48]: extraneous input ')' expecting ';'
        System.out.println(new BBS().Start(10 iji)));
                                               ^
```

图 7: 错误修复机制

3) 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）。

```
ParserRuleContext ctx = parser.goal();
generateAST(ctx, false, 0);
```

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）。

```
//ignore unnecessary rule context nodes
boolean ignored = ctx.getChildCount() == 1 && ctx.getChild( i: 0) instanceof ParserRuleContext;
if (!ignored) {
    String ruleName = MiniJavaParser.ruleNames[ctx.getRuleIndex()];
    if(verbose)//to display original text or not
        System.out.println(indentation+"[label=\""+ruleName+"\n"+ctx.getText()+"\"]");
    else
        System.out.println(indentation+"[label=\""+ruleName+"\"]");
    if(!preIndentation.equals("")) {//exclude root node's connection
        System.out.println(preIndentation+"->"+indentation);
    }
}
```

图 9: 非叶子结点的打印过程

```
for (int i = 0; i < ctx.getChildCount(); i++) {
    ParseTree element = ctx.getChild(i);
    //switch among different context
    if (element instanceof RuleContext) {//non-leaf nodes for parser rules
        generateAST((RuleContext) element, verbose,  indentation: indentation+tempChildCnt,indentation);
        tempChildCnt++;
    }
    else if(element instanceof TerminalNodeImpl){//leaf nodes for lexer rules
        Token t = ((TerminalNodeImpl) element).getSymbol();
        //to exclude literal names, only include symbolic names
        String lexerName = MiniJavaLexer.VOCABULARY.getSymbolicName(t.getType());
        if(lexerName!=null) {
            System.out.println(indentation+tempChildCnt+"[label=\""+lexerName+"\n"+element.getText()+"\"]");
            System.out.println(indentation+"->"+(indentation+tempChildCnt));
            tempChildCnt++;
        }
    }
}
```

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）。

```
please input your verbose choice: 1 for true, 0 for false
0
verbose is set to false
```

图 11: 请求用户输入 verbose 布尔值

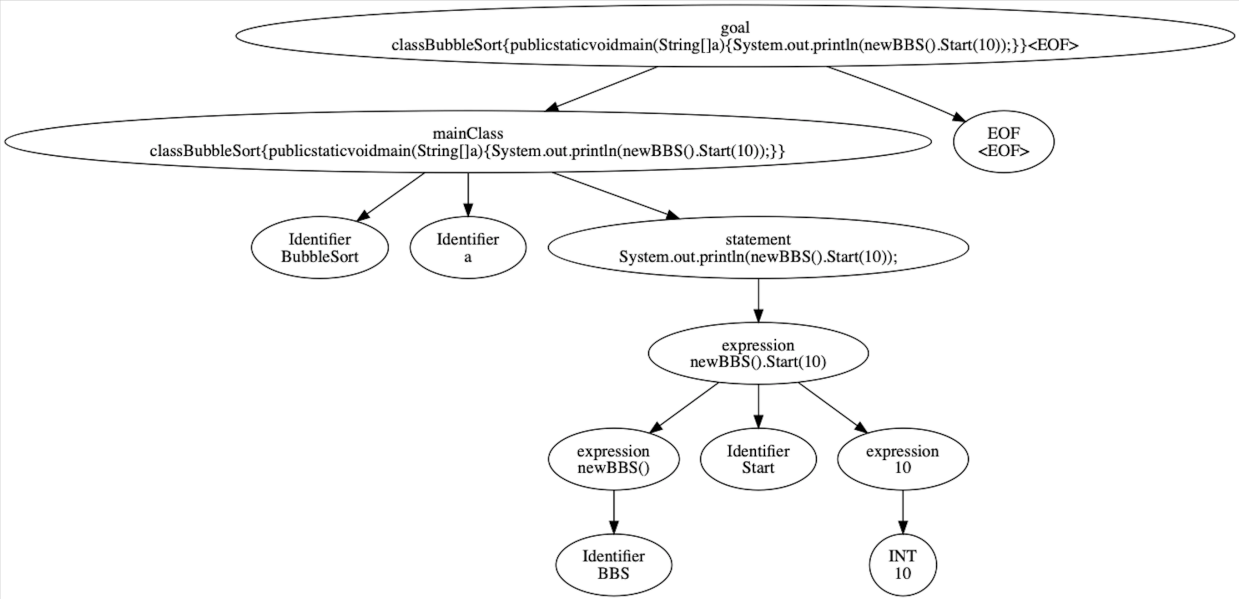

图 12: verbose 置为 true 的语法树效果图

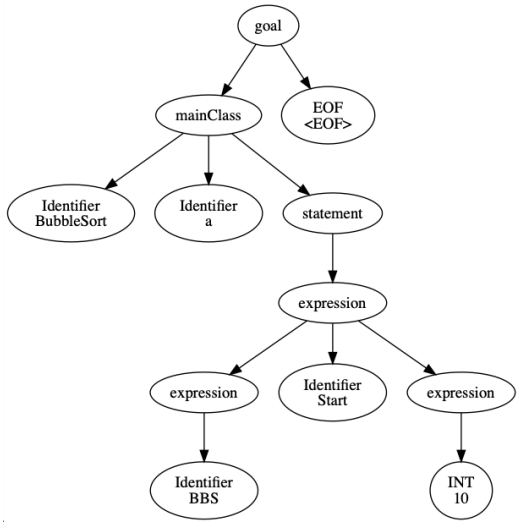

图 13: verbose 置为 false 的语法树效果图

而画图使用的是 graphviz[3]第三方软件接口，也就是按照需求生成 dot 语言[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。

```
String command = "src/draw/dot -Tpng -o src/graph.png src/draw/test.dot";
String openCmd = "open src/graph.png";
Process p = Runtime.getRuntime().exec(command);
```

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.

```
private static void generateAST(RuleContext ctx, boolean verbose, String indentation, String preIndentation)
```

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

[3] http://www.graphviz.org
[4] https://graphviz.gitlab.io/_pages/doc/info/lang.html

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）。

```
expression:
            '(' expression ')'                                                  #parentheses
            | '!' expression                                                    #negation
            |expression '&&' expression                                         #and
            |expression '<' expression                                          #smaller
            |expression '*' expression                                          #multiplication
            |expression '+' expression                                          #addition
            |expression '-' expression                                          #subtraction
            | expression '[' expression ']'                                     #readMemory
            | expression '.' 'length'                                           #length
            | expression '.' Identifier '(' (expression (',' expression)*)? ')' #callFunc
            | INT                                                               #integer
            | 'true'                                                            #true
            | 'false'                                                           #false
            | 'this'                                                            #this
            | Identifier                                                        #identifier
            | 'new' 'int' '[' expression ']'                                    #newIntArray
            | 'new' Identifier '(' ')'                                          #newFunc
            ;
```

图 16: expression 的匹配规则的正确顺序

4. 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。通过动手实践，先理解这个过程需要的前提、处理的步骤和最后怎样利用这些信息画出最后的抽象语法树。理解了这些之后，我们先创建相关的语法词法定义，编译相关的文件；然后通过相关的 label 定义，递归遍历 Parser 树处理叶子结点和非叶子结点；最后使用 Graphiz 画出抽象语法树。在这个过程中我们深刻认识到由自然语言到机器语言的过程远比我们想像的复杂，更好地认识了编译器的原理。

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。