编译原理 项目报告

一、 编译原理简介

编译，是利用编译程序从源语言程序转换成目标语言的一个过程。其主要包括前端和后端两个部分，前端用于将源语言程序翻译成中间表示树(IR 树)，主要包括了词法分析、语法分析、语义分析、栈帧布局等动作，后端用于将中间表示树转译成目标语言代码(常常为汇编语言)，包括了规范化、指令选择、控制流分析、数据流分析、寄存器分配等操作。

本项目注重实现编译器的前端，识别 MiniJava 语言，以抽象语法树为输出，主要包括了词法分析、语法分析、语义分析。其中词法/语法分析主要由词法/语法自动生成工具 ANTLR 完成，语义分析包括了变量声明检查、类型检查等。

词法分析，主要是用于对程序中每一个词的识别，例如 ID，数字，保留字等。词法分析可以通过正则表达式来表示，用有限状态自动机实现。

语法分析，主要是对程序中每句话语法的判别，通常采用上下文无关文法，利用下推自动机进行实现。在语法分析时，为了避免二义性，会对文法进行大量的等价变动，一个合理的程序设计语言，通常会有一个 LALR(1)文法，LALR(1)文法基本是程序设计语言及语法分析生成器的标准。

语义分析，则是判断程序的语义正确性，包括了各类变量是否已经定义过、变量类型是否匹配、方法的参数列表是否匹配等等。

在以上各个阶段，都会遇到相应的编译错误。例如 token 无法识别(词法错误)，左右括号不匹配(语法错误)，表达式类型不匹配(语义错误)等。其中词法错误和语法错误直接由定义出的文法决定，因而词法生成器和语法生成器可以处理这样的

错误，而语义错误则取决于语言本身的特性，因而需要自己去实现，也是本项目的重点部分。

通过这几步之后，我们可以得到源程序的一棵抽象语法树。抽象语法树将源代码表示成树的形式，以便于后面的处理。

二、 ANTLR 工具和文法的定义

词法语法分析器生成器有很多，Flex/Bison，Lex/Yacc，Jlex/CUP，JavaCC，ANTLR 等，在这里，我选择了 ANTLR。主要是基于：

1. ANTLR 有自己的主页和详细的文档，上手容易；
2. ANTLR 有一本配套的，非常详尽的 Reference；
3. ANTLR 的文法定义非常友好，且输出方式多样化，支持 tokens, tree, GUI 等多种输出方式，交互良好；
4. TA 及其它同学的推荐

```
grammar MiniJava;

prog    : mainClass (classDec)* EOF;

// parser rules
mainClass: 'class' ID '{' PUBLIC STATIC VOID MAIN '(' STRING '[' ']' ID ')' '{' statement '}' '}';
classDec : 'class' ID (EXTENDS ID)? '{' (varDec)* (methodDec)* '}';
varDec   : type ID ';';
methodDec: PUBLIC type ID '(' (type ID (',' type ID)*)? ')' '{' (varDec)* (statement)* RETURN expr ';' '}';
type    : INT '[' ']'       #IntArrayType
        | BOOLEAN           #BooleanType
        | INT               #IntType
        | ID                #ClassType
        ;
statement: '{' (statement)* '}'                         #BraceStatement
         | IF '(' expr ')' statement ELSE statement     #IfStatement
         | WHILE '(' expr ')' statement                 #WhileStatement
         | 'System.out.println' '(' expr ')' ';'        #PrintStatement
         | ID '=' expr ';'                              #AssignStatement
         | ID '[' expr ']' '=' expr ';'                 #AssignArrayStatement
         ;
expr    : expr '&&' expr                                #AndExpr
        | expr '<' expr                                 #LessExpr
        | expr ('+' | '-') expr                         #PlusMinusExpr
        | expr '*' expr                                 #TimesExpr
        | expr '[' expr ']'                             #IndexExpr
        | expr '.' LENGTH                               #LengthExpr
        | expr '.' ID '(' (expr (',' expr)*)? ')'       #CallExpr
        | INT_VAL                   #IntExpr
        | TRUE                      #TrueExpr
        | FALSE                     #FalseExpr
        | ID                        #ClassExpr
        | THIS                      #ThisExpr
        | NEW INT '[' expr ']'      #NewIntArrayExpr
        | NEW ID '(' ')'            #NewClassInstanceExpr
        | '!' expr                  #NotExpr
        | '(' expr ')'              #ParenthesisExpr
        ;

```

代码段 1 MiniJava 语言的文法定义(parse rule)

ANTLR 的文法定义并不复杂，主要分了两部分，parser rule 和 lexer rule。需要注意的是区分这两种 rule 的方式是大小写，小写开头的 rule 是 parser rule，大写开头的则是 lexer rule。匹配规则时，按照从上往下的顺序进行匹配，匹配第一个遇到的合法规则。

ANTLR 还有一些常常需要使用到的 trick：比如说对于细分的文法，可以通过加#表示 tag，在生成 parser 时生成器会对该 rule 的每一条细分文法都进行区分，对于 statement/expression 的定义非常有必要。此外，fragment 标记可以标出那些只用于 grammar 本身的中间量，例如 Digit, Letter 等，用于更简洁明了地去表示 ID 等真正的 rule，而不产生对应的访问方法。

```
InputStream in = new FileInputStream(inputFile);
System.out.println("File loaded.");
ANTLRInputStream input = new ANTLRInputStream(in);
MiniJavaLexer lexer = new MiniJavaLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
MiniJavaParser parser = new MiniJavaParser(tokens);
ParseTree tree = parser.prog();

// Check symbols
ParseTreeWalker walker = new ParseTreeWalker();
DefPhase defPhase = new DefPhase();
walker.walk(defPhase, tree);
RefPhase refPhase = new RefPhase(defPhase.getGlobalScope(), defPhase.getScopes(),
walker.walk(refPhase, tree);

// Show AST in console
System.out.println(tree.toStringTree(parser));

// Show AST in GUI
JFrame frame = new JFrame("ANTLR AST");
JPanel panel = new JPanel();
TreeViewer viewer = new TreeViewer(Arrays.asList(parser.getRuleNames()), tree);
viewer.setScale(1.0);
panel.add(viewer);
frame.add(panel);
frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
frame.setSize(800, 400);
frame.setVisible(true);
```

代码段 2 编译器主程序

ANTLR 生成之后，主要包括了以下 Java 文件：*.tokens 文件对遇到的关键词进行编号，*Lexer.java, *Parser.java 则是对应的词法分析器和语法分析器，*Listener.java 实现了一个访问语法树的接口,而*BaseListener.java 则是这个接口

的一个基本实现。我们需要继承这个 BaseListener，继续完成我们需要的语义检查的工作。

这是主程序的代码框架，我们从文件输入流读取源程序，先进行词法分析(lexer)得到了对应的 tokens，然后将 tokens 作为输入，得到相应的 parser，由这个 parser 建立出一棵语法树。这之中，DefPhase 和 RefPhase 就是两个继承自 MiniJavaBaseListener 的类，用于遍历语法树。

接来下的部分是对这棵语法树进行两次遍历，进行语义检查。最后在 console 和 GUI 中分别输出相应的语法树结果。

三、 源代码的架构和工作原理

在语义检查里，核心的一块在于符号(Symbol)的检测，这包括了作用域检查(Scope)，声明检查，表达式类型匹配等。

自定义了 2 个枚举类型(SymbolType, VarType), 4 个 Symbol 类(Symbol, VarSymbol, MethodSymbol, ClassSymbol)，一个 Scope 接口和对应的 Block 类。下图即为这些自定义的类、接口、枚举类型的示意图。关于该图及对应的一些细节的解释如下：

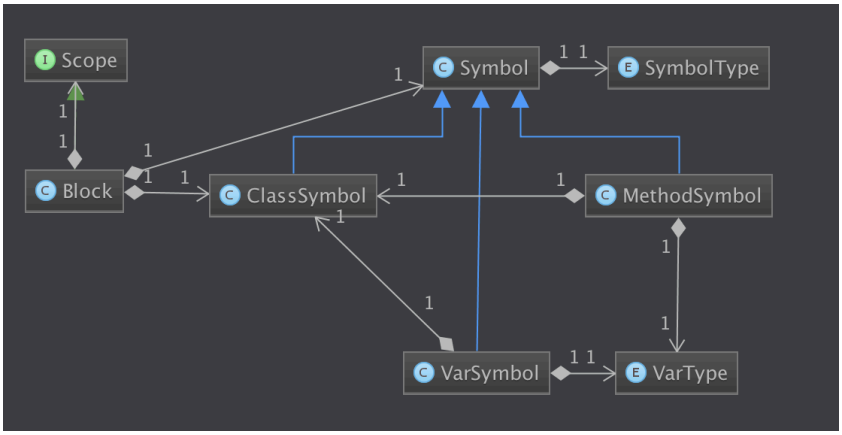

图 1 类图及依赖关系简图(由 Intellij 生成)

1. Symbol 表示了一个符号，其对应的 SymbolType 有三种：类、方法、变量或实

例，分别由三个子类继承。Symbol 具有 name 和 symbolType 两个基本属性。

2. Scope 定义了一个作用域的接口，Block 实现了该接口。Block 内以 Map 的形式存放了当前作用域里的符号表 symbols，因而依赖于 Symbol 类。同时 Block 是一个树状结构，除了全局作用域 globalScope 以外，其实例均有 outerScope 和 outerClass 属性，outerScope 用于存放上一层 Scope，outerClass 则是针对 THIS 关键字而写，保存里该 Scope 所在的类，在构造一个 Block 时会自动拷贝一份其 outerClass 属性(除全局 Scope 外)。
3. 对于一个 Symbol 的检测，是在当前 Scope 里调用 lookup 方法。该方法检测当前 Scope 附带的符号表 symbols 里是否含有该符号。若没有，则依据 outerScope 向上逐层查找符号表，直到最外层也没找到，或找到了对应的符号。找到的符号会依据其 symbolType，判断类型是否匹配，作出对应的动作。
4. ClassSymbol 表示了一个类的符号，没有额外的属性。

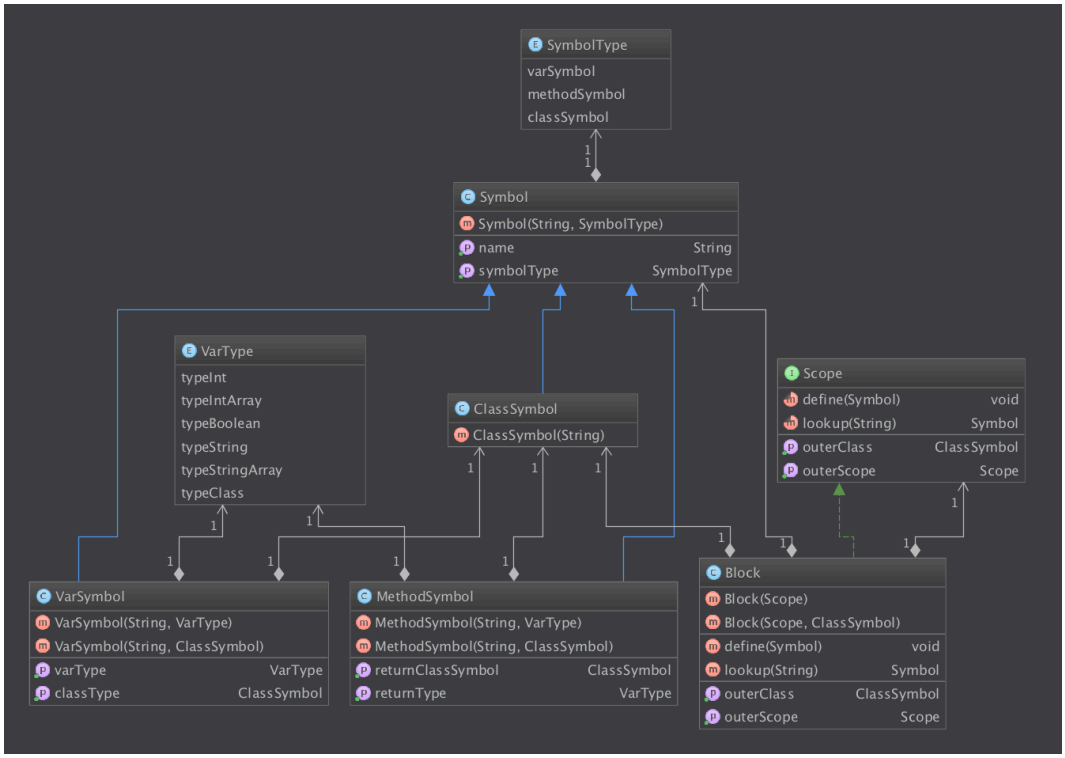

图 2 类图及依赖关系详图（包括属性，方法，构造器）

5. VarSymbol 表示了一个变量的符号，变量有对应的 VarType。一个变量可能

是某个 Class 的实例，因而还需要存储对应的 ClassSymbol。

6. MethodSymbol 表示了方法的符号，方法的返回类型由 VarType 存储。同样，方法返回类型可能是一个类，此时同样需要存储这个类的 ClassSymbol。

以上是需要用到的类的介绍，在类型检查的实现过程中，进行了两遍 pass，这是为了支持向后引用（在前面引用后面定义的类、方法）。经推理，两遍还不能支持方法返回类型的向后引用，若要支持方法返回类型向后引用，则需要至少三遍。

由于类名和方法名允许向后引用，而变量必须先申明再使用，因而第一遍只定义 ClassSymbol 和 MethodSymbol，而将所有的 VarSymbol 留在第二遍定义。

四、 错误处理机制

ANTLR 的生成器可检测词法错误和语法错误，且能够进行一个 token 范围的错误修复，如果需要对这里的错误报告进行重写，可以继承 ErrorListener 自己实现，并替代原先的 ErrorListener。由于 ANTLR 原先的错误报告机制已经比较完善，本项目没有对原先部分进行继承重写，而是增加了语义错误的检查。

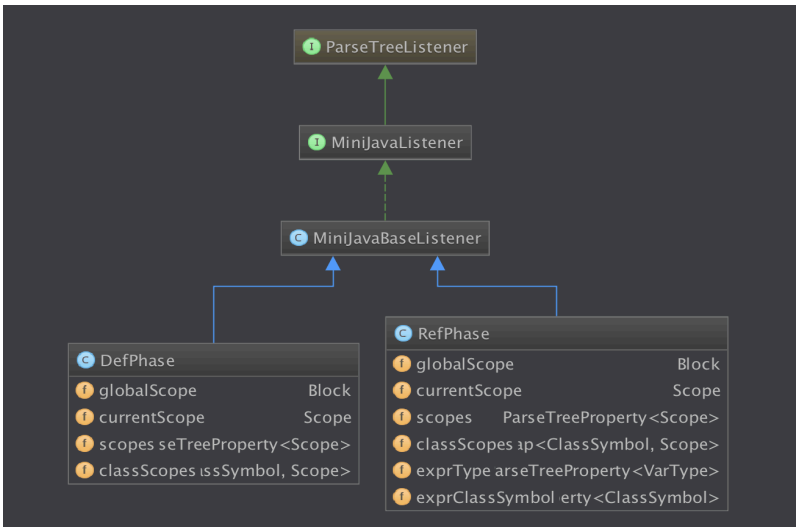

图 3 DefPhase 和 RefPhase 的属性

错误处理机制采用遍历两遍语法树的形式，分别命名为 DefPhase 和 RefPhase，

第一遍用于定义类和方法，第二遍用于进行详细的错误检查。这两个类都继承自 MiniJavaBaseListener 类，为每个 parser rule(如果子规则标记了 tag 则是每个 tag)都自动生成了 enter 和 exit 方法，在这条语法规则被匹配的开始和结束时调用。

DefPhase 拥有 globalScope, currentScope, scopes, classScopes 四个变量。除了 currentScope 用于读取代码时的作用域控制外，另三个变量均会作为参数传递给 RefPhase 的构造器。其中比较特殊的是 classScopes 这个变量，它是为了解决在一个类的 Scope 里调用另一个类的方法(这一问题不能直接由该 Scope 里搜索 Symbol 来解决)，没有这个变量连样例代码都不能通过编译器测试。

RefPhase 则多了 exprType 和 exprClassSymbol 两个 Map，前者用于保存每个 context(语法树中的结点)的变量类型，以进行表达式计算时的类型检测，后者则是前者的补充，当某个节点的变量类型是一个类时，保存其对应的 ClassSymbol。

```
1  class Factorial{
2      public static void main(String[] a){
3          System.out.println(new Fac().ComputeFac(10));
4      }
5  }
6
7  class Fac extends F{    // Error
8      public int ComputeFac(int num){
9          int num_aux;
10         n = 1;          // Error
11         num_aux = true; // Error
12         num_aux = num.length;   // Error
13         if (num < 1)
14             num_aux = 1;
15         else
16             num_aux = num * (this.ComputeFac(num-1));
17         num = this.Compute(num-1);  // Error
18         num = num_aux.ComputeFac(num-1);    // Error
19         num = num + true;       // Error
20         return num_aux ;
21     }
22 }
```

代码段 3 一段 MiniJava 代码，Intellij IDE 已经识别出了其中错误的地方

由于错误检查非常繁多细致，对每个语句每个表达式都需要检测，一处崩溃又可能导致整个编译器崩溃，下面通过一个实际例子，对实现的常用几类语义错误进行详解和演示。

1. 符号未声明或类型(Class, Variable, Method)不匹配

例如第 10 行，变量 n 此前没有定义过

例如第 17 行，调用了一个不存在的方法

2. 被继承类不存在

例如第 7 行，Fac 继承了一个不存在的类。这里采用了错误恢复，即使不存在也不影响后续处理。

3. 表达式运算不合法(索引不是整数，对非布尔类型使用布尔运算，对非整数类型使用数字运算，赋值号左右类型不匹配)

例如第 11 行，左右类型不匹配

例如第 19 行，将整型变量和布尔变量进行数值运算

4. 方法调用不合法(对非 Array 类型求 length，对非实例的符号调用方法)

例如第 12 行，对一个非 Array 类型的变量求 length

例如第 18 行，对一个非类名或 THIS 关键词调用了方法

```
File loaded.
line 7:18 Inherited class F doesn't exist.
line 10:2 Symbol n doesn't exist.
line 11:2 Symbol num_aux is not matched with expression.
line 12:12 num is not an array of int/string
line 17:13 The method Compute doesn't exist or is not a method symbol.
line 18:8 call error: The class num_aux doesn't exist.
line 19:14 The +/- operator must be operated on two int expression.
(prog (mainClass class Factorial { public static void main ( String [ ] a )
```

代码段 4 以上样例错误的输出信息，能够正确检测出语义错误并输出语法树

5. 未知错误

未知错误主要是由于本身存在语法错误导致的。由于实现的编译器在类型检查时默认了语法正确，而这样的错误容易导致编译器内部崩溃报错(取子节点时不存在)，因而需要在错误处理时加上 try…catch…语句，针对一些未考虑到的意外情况，也能拥有一定的容错能力。同时也避免了编译器本身报错。

撰写的语义错误检查器，在网上的 8 个正确的 MiniJava 样例程序中，可通过 7 个，最后一个未通过的原因是使用了被继承类的变量(而误被认为是未定义)。

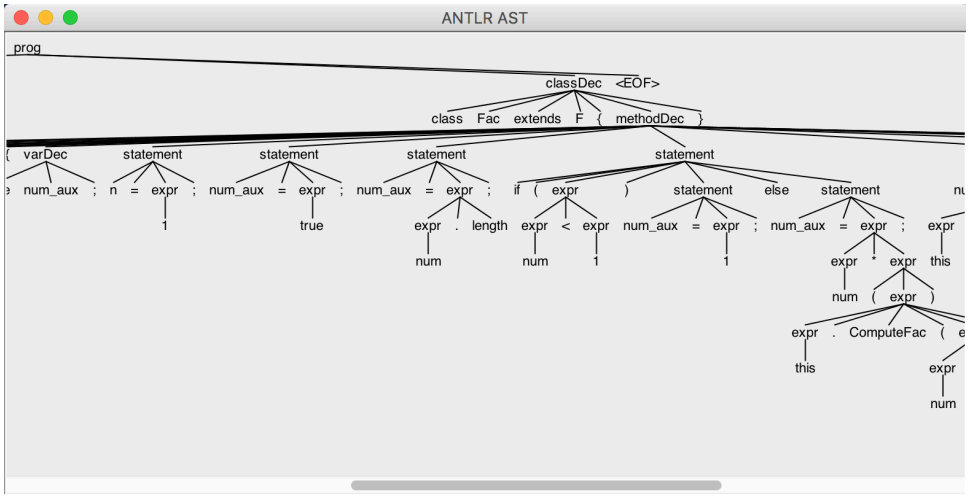

图 4 以上样例的抽象语法生成树(AST)局部

五、 错误恢复

在错误处理时，常常因为一个错误，链式地引发了一系列错误。因而我们需要做一定程度的错误恢复。

一类是上述所说的由语法错误导致的，这类错误会导致不能正确得到语法树。对于这样的错误，其方法即为在语法规则里容错。

另一类则是语义错误的恢复。一种情况时继承了一个无效的类，这个比较容易处理，忽略即可。另一种则是由于某个表达式不合法，导致其外层的每一个表达式均不合法，引发一连串的错误，这可能不是一种合理的解决方案。

例如表达式!(a == b) && (b > c)，如果误将 a == b 写成了 a = b，则会引发一系列的问题。我的处理方式是，对于某些仅针对特定类型的表达式计算，例如!, &&, >等，在遇到了某个错误时，输出错误信息，但返回理应得到的正确的类型，以供上一层调用。例如!(a = b)虽然遇到了错误，但是会认为这个节点是 boolean 类型的，参与下一次的运算。再如在处理 a.length 这样的语句时，无论是否报错，都将整个节点认为是 int 类型。

六、 遇到的问题和解决

1. This 关键词

This 关键词特别的地方在于它没有显式指出类的符号，因而不能直接调用 lookup 方法查找，但是事实上不用查找因为它一定在某个类里调用。处理方法是对于每个 Scope，加上 outerClass 属性，直接保存其所属的类。

2. 调用另一个类的方法

任何一段样例代码中，MainClass 都会调用后面 Class 的方法，这带来了两个问题。第一是该类还没有定义，第二则是该类的方法压根不在当前的作用域内。对于第一个问题，我将一遍扫描改成了两遍(见原理部分)，而对于第二个问题，则是在第一遍定义类和方法时，保存了一个<ClassSymbol, Scope>的 Map，以供直接调用另一个类的方法。

3. 继承的变量处理

一个被报错的正确样例程序是 TreeVistor，由于其类中调用了被继承而来的变量，因而被我的编译器误认为没有定义就使用。这个问题没有解决，目前的思路是说要对于类的继承关系，同样维护一个树状的符号表。

七、 项目感想

编译原理是大学的最后一门主要的专业必修课。纸上得来终觉浅，绝知此事要躬行。通过对编译器前端的实现，加深了对编译原理的理解，了解了编译器实现词法分析、语法分析、语义分析，以及各类错误报告和错误恢复的大致流程。

整个项目独立完成，主要的类除了参考 ANTLR Reference 外，均由自己设计完成，在项目架构的设计上思考了很多，完成度也比较满意。

编译器能够完成的上限非常高，继续做下去可以考虑继承、重载，可以实现更完善的语言语法和语言特性。总之乐趣无穷，是一个能够学到很多东西、非常有意思的 Project。