

MiniJava 编译器前端——基于 ANTLR 的词法语法分析

《编译原理》课程项目文档

郑哲东 12307130083@fudan.edu.cn

韩韵衡 12307130140@fudan.edu.cn

一、概述

ANTLR (Another Tool for Language Recognition) 是一个功能强大语言识别工具，因其语法简洁、运行高效而被广泛应用。为实现 MiniJava 语言的编译器前端，我们选用 ANTLR 作为主要的词法和语法分析工具。

输入 MiniJava 源程序后，我们首先编写语法文件，使用 ANTLR 分析代码，得到其语法分析树 (Parse Tree, Concrete Syntax Tree) 和抽象语法树 (Abstract Syntax Tree)。在错误检查和提示方面，我们除去输出 ANTLR 分析过程中检出的默认错误外，我们还利用 ANTLR 支持的内嵌语句 (actions) 补充了其他常见的语法错误。

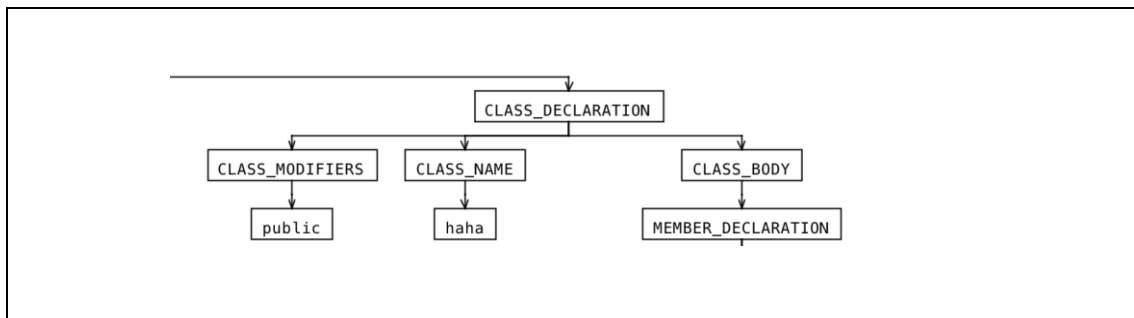
在项目实现过程中，我们先后使用了 ANTLR 3 和 ANTLR 4 两个版本，程序运行环境为 Java 1.7。项目文件中包含了语法文件 (MiniJava.g4)、测试程序 (testRig.java) 和 6 个测试用例。下面我们首先说明语法文件的构成，然后介绍具体的实验过程。

二、语法和代码解释

我们从类 (class) 声明开始，中间经由分支循环语句 (if, while)，直到基本的表达式和变量声明——自顶向下、逐步细分来编写 ANTLR 所需的语法文件。首先是最基础的程序块 (compilationUnit) 声明，其中包含了主类 (mainclass) 和其他类 (classOrInterfaceDeclaration) 的两部分。注意到如果 Java 程序中只能包含一个主类 (public class)，所以在这里嵌入了输出语句来提示主类相关错误。

接下来是主类 (mainclass) 和其他非主类 (classOrInterfaceDeclaration) 的声明。主类必须以 “public” 作为其限定符，而其他内容和非主类并无区别，因此两者共享类主体语法 (classDeclaration)，但非主类只能使用除去 “public” 的其他限定符 (classOrInterfaceModifiers)。此外为了输出抽象语法树，还应配合使用 “->” 和 “^” 操作符来输出指定的语法树分支节点和节点文本内容。

```
compilationUnit
    :classOrInterfaceDeclaration* mainclass? compilation?
    ;
compilation
    :classOrInterfaceDeclaration+ compilation?
    |mainclass+ classOrInterfaceDeclaration* {System.out.println("error:multi_main_class");}
    ;
mainclass
    : 'public' classDeclaration -> ^(MAIN_CLASS classDeclaration)
    ;
classOrInterfaceDeclaration
    : classOrInterfaceModifiers classDeclaration
    -> ^(CLASS_DECLARATION classOrInterfaceModifiers classDeclaration)
    ;
```

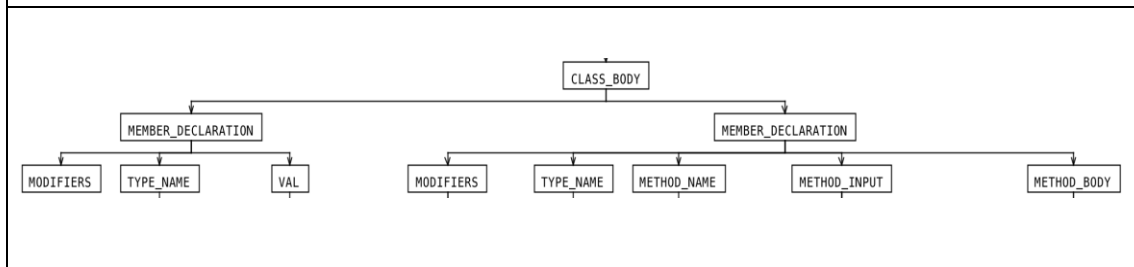


为叙述简洁，语法文件中细节声明如类主体（`classBodyDeclaration`）等内容不展开描述。接下来是类成员（`memberDecl`），类内成员主要有构造器（`genericMethodOrConstructorDecl`）、成员变量、成员函数（`memberDeclaration`）和内部类（`classDeclaration`）四种。其中构造器的名称必须和类名一致，否则使用嵌入语句报错。除去没有返回值类型外，构造器的声明和成员函数无异，以下会解释函数声明，故不再赘述构造器细节语法。

```

memberDecl
  : genericMethodOrConstructorDecl
  | memberDeclaration
  | Identifier constructorDeclaratorRest
  | classDeclaration
  ;

memberDeclaration
  : mtypename=mtypename method_name=Identifier(' ' formalParameterDecls? ' ') methodBody
  -> ^(TYPE_NAME $mtypename) ^(METHOD_NAME $method_name)
    ^(METHOD_INPUT formalParameterDecls?) ^(METHOD_BODY methodBody)
  | typename = type fieldDeclaration
  -> ^(TYPE_NAME $typename)^(VAL fieldDeclaration)
  ;
  
```



成员函数具体分为返回值类型（`mtypename`）、函数名（`Identifier`）、参数声明（`formalParameterDecls`）和函数主体（`methodBody`）四个部分，其中函数主体将转到具体的循环分支语句块（`statement, block`）再作详细分析。而成员变量声明分为两部分，即变量类型（`type`）和（可能多个）变量名（`fieldDeclaration`）。使用左递归将多个变量名拆解为单个变量后，便可再对变量初始化（`variableInitializer`）作详细分析，最终在抽象语法树中一个变量声明语句被输出为三个部分，即限定符、变量名和初始化语句（可以省略）。

```

statement
  :block
  |'if' parExpression statement (options {k=1;}: 'else' else_statement)?
  ->^(IF_BLOCK ^(CONTROL parExpression) ^(THEN statement) ^(ELSE else_statement?))
  |'for' '(' forControl ')' statement
  ->^(FOR_BLOCK ^(CONTROL forControl) ^(BODY statement))
  
```

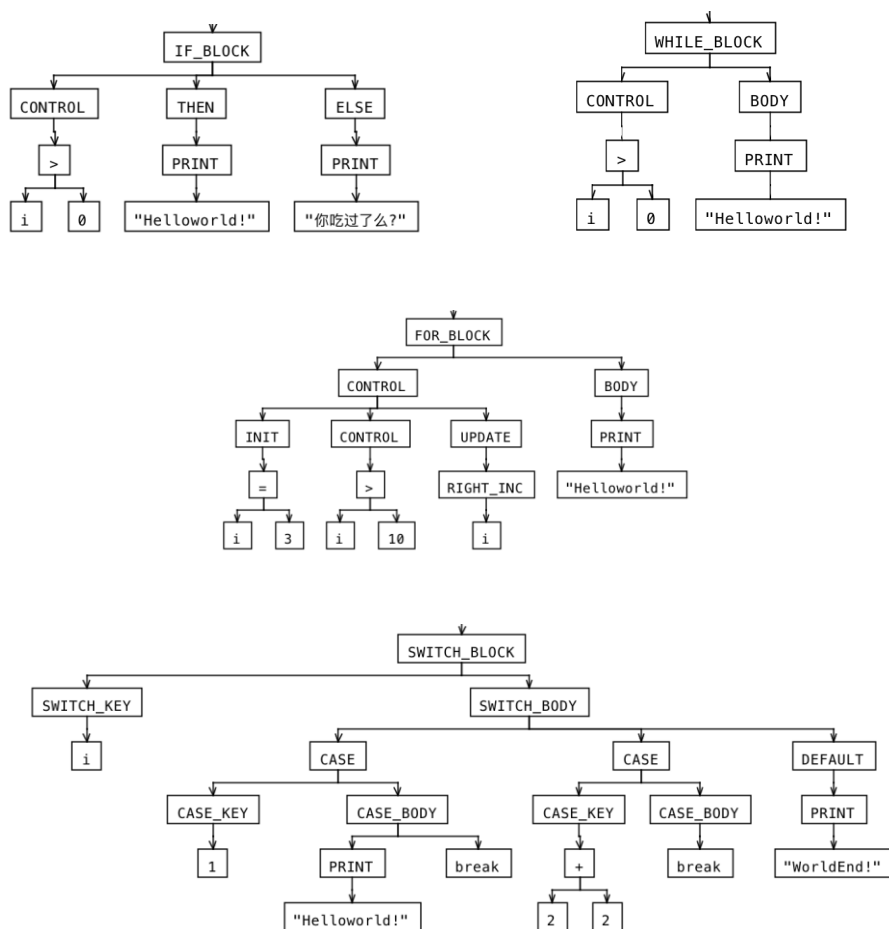
```

|'while' parExpression statement
->^(WHILE_BLOCK ^(CONTROL parExpression) ^(BODY statement))

|'switch' parExpression '{' switchBlockStatementGroups '}'
->^(SWITCH_BLOCK ^(SWITCH_KEY parExpression)
^(SWITCH_BODY switchBlockStatementGroups) )

|'return'^ expression? ';'!
|'break' ';'!
|'continue' ';'!
|';'
|statementExpression ';'!
;

```



具体的语句块分析较为复杂，主要分为分支（if, switch）、循环（for, while）和其他语句（控制语句 return, break, continue、表达式 statementExpression）三种。If 语句包含三个部分，即控制条件、then 部分和 else 部分。控制条件为一个布尔表达式（parExpression），then 和 else 部分均为语句块（statement），注意 else 部分可以缺省。同理，for 语句分为两个部分，即循环控制块（forControl）和循环主体（statement），其中循环控制块会继续细分为三个部分处理；而 while 语句分为控制条件（parExpression）和循环体（statement）；switch 语句分为跳转条件（parExpression）和各项分支（switchBlockStatementGroups），其中各项分

支部分也会继续细分。

最后是基本的表达式，根据表达式类型，即赋值语句、单双多目运算分别递归定义。代码较繁琐但是内容单一，故仅取几例作解释说明。如下表所示，逗号表达式为双目运算，但可以连续使用，故需要使用“*”限定符号和递归来定义多个表达式用逗号串联的情况；而三目运算符包含了条件和两个语句块，其间用“:”和“?”隔开；其余运算多为双目运算，如“||”运算，和逗号表达式是类似的。在抽象语法树中，表达式的运算符在树根处，而其儿子就是参与运算的多个子表达式。例如，二元运算加法在树根处为加号，而两个儿子为所加和的两个加数；同理三目运算符的表达式树则可以有三个儿子。

<pre>expressionList :expression (',' expression)* ; conditionalExpression :conditionalOrExpression ('?' conditionalExpression ':' conditionalExpression)? ; conditionalOrExpression :conditionalAndExpression (' ' conditionalAndExpression)* ;</pre>	<pre>graph TD LOC_VAL[LOC_VAL] --> int[int] LOC_VAL --> EQ[=] LOC_VAL --> return[return] int --> i[i] EQ --> 1[1] EQ --> star[*] star --> plus[+] star --> minus[-] plus --> x1[x] plus --> y1[y] minus --> x2[x] minus --> y2[y] return --> 0[0]</pre>
--	--

和标准的 Java 语言相比，我们没有实现 do-while 语句的分析，这是因为 for 循环和 while 循环已经足以支持循环语句的全部需求，而 do-while 语句并没有实现新的功能，因此可以省去。另外为了语法简洁、实现方便，接口（interface）和继承（extends）也没有实现，因为它们的语法规则较为复杂，因此错误检测和修复也更加困难。

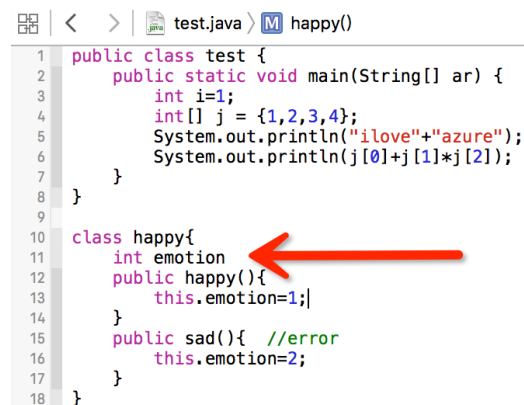
而和项目要求中给定的基本 MiniJava 语法规则相比，我们附加实现了常用的 for 循环和 switch 语句，还有对于基本类型、字符串类型和数组类型的支持，以及对于 ANTLR 没有判定但是常见的语法错误的检测和提示。综上所述，我们实现的 MiniJava 编译器前端支持较为完备的类、变量函数声明、循环分支和表达式等语句语法。

三、 实验

首先我们对一个简单的测试用例分别构建语法分析树和抽象语法树，很容易看出其中的区别。对于同样一段程序，具体分析树对文本逐步细分，其中分支节点代表了程序的不同语句块，而叶子则是源程序中的原始字符。具体的语法分析树由语法分析直接产生，代表了语法分析的全过程，但是其中包含了大量的冗余分支节点——比如以“statement”为根的子树代表了一个语句块，但是语句块的具体内容在子树的叶子上才出现——几乎所有的分支节点都是如此。另外很多格式符号如“[”、“]”和“;”也在叶子上出现，而这些符号对于明确表示语法树的结构是没有帮助的，因此这些叶子节点也是冗余的。

抽象语法树和具体语法分析树不同，它不包含具体的源程序字符（如多余的“[”、“]”和“;”符号和“statement”分支），取而代之的是确实表征了源码结构和内容的节点。如下

为必要的语法成分缺失，如语句结束缺失分号、函数没有返回类型等。在 ANTLR 3 中，这种错误通过 `Exception` 抛出，在 ANTLR 4 中则更加方便，会直接提示错误在源码中的位置和所需求的缺失成分，如下图所示。

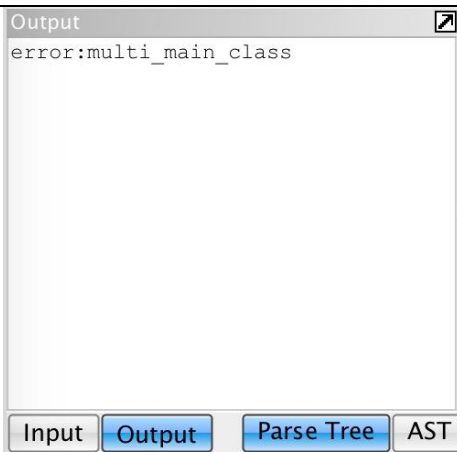


```
1 public class test {
2     public static void main(String[] ar) {
3         int i=1;
4         int[] j = {1,2,3,4};
5         System.out.println("i love"+"azure");
6         System.out.println(j[0]+j[1]*j[2]);
7     }
8 }
9
10 class happy{
11     int emotion
12     public happy(){
13         this.emotion=1;
14     }
15     public sad(){ //error
16         this.emotion=2;
17     }
18 }
```

```
→ mini-java git:(master) ✕ grun Minijava goal -gui <test.java
line 12:4 no viable alternative at input 'intemotionpublic'
```

而和语义相关的错误，ANTLR 本身是不支持检查的，所以我们通过嵌入 `action` 语句来判断和提示这类错误，例如下述程序中，一个 `java` 程序包含了超过一个主类（`public class`），则需要使用 `action` 语句打印错误为“`error:multi_main_class`”。

```
compilationUnit
    :classOrInterfaceDeclaration* mainclass? compilation?
    ;
compilation
    :classOrInterfaceDeclaration+ compilation?
    |mainclass+ classOrInterfaceDeclaration* {System.out.println("error:multi_main_class");}
    ;
```



Output
error:multi_main_class

Input Output Parse Tree AST

```
public class test1 {
    public static void main(String[] args) {
        return 0;
    }
}
class test3{
}
public class test5{
}
```

四、 过程和体会

本次编译课程的项目由于使用了 ANTLR 工具，其语法文件的规则简洁清晰，嵌入的 `action` 语句功能强大却易于编写，所以在实现上不是很困难。根据语法规则编写语法文件并嵌入 `action` 语句来检错报错是核心的内容，虽然所实现的 MiniJava 编译器前端并没有完整的功能，但是这对于理解词法语法分析过程、语法树的结构和构造、具体分析树和抽象语法树的区别有很大的帮助。

程序编译经历了长时间的发展已经日趋成熟，如今我们已经有了稳定高效的编译器，易用的 IDE，早已不需要自己实现这类工具。尽管如此，我们还是需要去了解编译程序过程中的算法和原理，去动手实现一点东西——这对于完善知识体系乃至今后处理各类文本数据都是很有裨益的。

韩韵衡 郑哲东
2015 年 12 月 28 日