编译 Project: Compiler for miniJava

14307130360 王丹青

一、 词法与句法分析

编译的主要工作是将一种高级语言（源语言）等价转换成低级语言（目的语言），从而使得机器可以执行。而在语言的翻译过程中，首先需要将源语言拆分开，识别出其充当的成分，分析成分之间的结构和含义，在将成分等价转化成目的语言成分，重新组合起来。词法分析就是拆分并且识别成分（token），句法分析就是分析成分之间的结构。

词法分析主要是根据该文法的构词规则来划分单词 token，并且在此过程中去除无用的成分，如空白符和注释。一般的分析工具是正则表达式和有限状态机。有限状态机作为正则表达式匹配的引擎，有两大类。分别是确定性有限状态机 DFA 和非确定性有限状态机 NFA。目前主流的正则表达式引擎是 NFA，相比 DFA 来说它的功能更加强大，特性丰富，但是存在最左子正则式优先匹配的缺点，有时会错过最佳匹配结果。

句法分析是根据上下文无关语法对输入的 token 流进行产生式规则的匹配。每个 token 匹配到对应的文法符号，并且由符号之间推导关系构成一棵语法树（parse tree）。语法树中父结点到子结点的连接揭示父结点推导出子结点的文法关系。句法分析能够判断输入的字符串是否符合该语言的文法结构。句法分析可以分成推导和规约两大类，即 LL 和 LR 两大类。它们一个是自上而下，一个是自下而上。

语义分析是初步判断语言的含义。句法分析生成的 parse tree 往往被称为具体语法树，它的结点是文法符号，因此对文法依赖程度较高。为了从一种文法转换成另一种文法，我们需要得到更加泛化的中间表示形式，与源语言和目的语言均无关。常见的中间代码表示方式由抽象语法树 AST，三元式，四元式等。在得到中间代码表示之后，往往需要进行初步的语义分析，即静态检查程序用法性错误。主要包括检查变量是否定义，类型是否正确等。

二、 工具比较与选择

Compiler Compiler（CC）即编译器的编译器。通过定义一种上下文无关文法，CC 通过对该文法的分析生成一个编译器，该编译器能够编译符合定义文法的程序。因为 CC 生成的程序是编译器，因此被称为编译器的编译器。

常见的 CC 程序有：Flex/Bison，Lex/Yacc，Jlex/CUP，JavaCC, ANTLR。

Lex 和 Yacc 是用 C 语言编写的，两者组成了一个具有词法分析和句法分析的 CC，其中 Lex 负责词法分析,生成一个 Lexer;Yacc 负责语法分析,生成一个 Parser。其中语法分析 Parser 使用 LALR(1)从下到上分析。

在 Lex 和 Yacc 的基础上，改进得到 Flex/Bison，Flex 对应 Lex，Bison 对应 Yacc。

Lex/Yacc 和 Flex/Bison 都是用 C 编写的，而 Jlex/CUP 是用 Java 编写的同一系列词法语法分析器。

JavaCC 是一个用 Java 编写的 CC，它包括了词法分析和语法分析两部分，默认使用基于自上而下的 LL(1)分析语法。相比 Lex/Yacc 和 Flex/Bison，更容易调试，并且有最为完善的错误诊断机制。此外它可以通过 JJDoc 工具将文法文件转化成可读性较高的 Html 文本。

Antlr 是一个用 Java 编写的 CC,同样包括了词法分析和语法分析两部分。它基于自上而下的 LL(k)来分析语法，比起 Lex/Yacc 的 LALR 更加强大，避免了 shift-reduce 和 reduce-reduce 之类的语法冲突。并且因为 LL(k)自上而下的特性，对于错误检测的支持更加友好，Anltr 本身提供了比较完善的错误提示和修复机制。取而代之的是，LL(k)会遇到左递归问题，Antlr4 对于左递归有一定容忍度，支持自身的左递归。

因为 LL 语法对于错误检测和修复更加友好，因此在工具选用中优先选择 JavaCC 和 Antlr。Antlr 由文法生成的词法，句法分析器代码结构更加清晰，默认使用的 LL(k)相比 JavaCC 的 LL(1)分析能力也更加强大，因此最后实验选用 Antlr。

三、 项目代码

3.1 源代码结构

```
COMPILER
├── SRC                    # 代码
│   ├── EXAMPLES           # MINIJAVA 样例
│   └── SRC/MAIN/JAVA      # 源代码
│       ├── MINIJAVA       # 编译器 JAVA 代码
│       │   ├── ANTLR      # 和 ANTLR 相关类及重载
│       │   │   ├── GEN    # ANTLR4 从 G4 文件中自动生成的 JAVA 代码
│       │   │   │   └── ...
│       │   │   ├── ASTTREE.JAVA              # AST 结点以及遍历算法
│       │   │   ├── MYERRORLISTENER.JAVA      # 从 ANTLR4 中重载的错误处理
│       │   │   ├── MYERRORSTRATEGY.JAVA      # 从 ANTLR4 中重载的错误处理策略
│       │   │   ├── MYMINIJAVAASTVISITOR.JAVA # 从 ANTLR4 中重载的 VISITOR 遍历
│       │   │   ├── SYMBOLENTRY.JAVA          # 符号表条目
│       │   │   └── SYMTABSCOPENODE.JAVA      # 符号表
│       │   ├── LISPTOTREE.JAVA    # 从 LISP 格式转化 TREE 的 SWING 绘图代码
│       │   ├── LISPTOTREEVIEW.JAVA # 封装 LISP 的树可视化格式的类
│       │   └── MAIN.JAVA          # 主程序
│       └── MYMINIJAVA.G4  # MINIJAVA 语法文件
├── README.MD          # README
└── REPORT.PDF         # 报告
```

3.2 核心代码

核心代码可以分成三部分：

- ✓ 词法和文法的定义
  - ✧ MyminiJava.g4
  - ✧ antlr4 生成的 miniJava/antlr/gen 文件夹
- ✓ 生成抽象语法树
  - ✧ 遍历生成 AST
    - ◆ ASTTree.java
    - ◆ MyminiJavaASTVisitor.java
  - ✧ 可视化
    - ◆ LISIPToTree.java
    - ◆ LIPSToTreeView.java
- ✓ 文法、句法和语义检查（见第四部分错误修复）
  - ✧ 文法句法检查
    - ◆ MyErrorListener.java
    - ◆ MyErrorStrategy.java
  - ✧ 语义检查
    - ◆ SymbolEntry.java
    - ◆ SymTabScopeNode.java

3.2.1 词法和文法定义

根据 miniJava 项目的 BNF 语法，可以得到简单的语法规则。在此基础上进行了以下处理：

a) Anltr4 的词法和句法规则可以写在一个 g4 文件中。其中词法规则 lexer rule 以大写字母开头，parser rule 以小写字母开头

b) 在进行句法分析的时候需要忽略空白符号和注释内容。空白符号直接采用 skip 跳过，而注释内容由于可能未来需要，因此使用 antlr4 的 hidden channel 保存。注释支持单行“//”和多行“/**/”模式，且支持嵌套注释，“/*”自动匹配最近的“*/”，多余的“*/”将被识别成错误 token。

c) 为了后续对错误提示和恢复更加友好，在 g4 文件中添加了常见错误匹配方式，比如多一个括号’{’ (statement)* ’}’ ’}’或者少一个括号’(’ ’(’ expression ’)’，使用 notifyErrorListeners 方式定制错误提示。

3.2.2 生成抽象语法树

（1）Antlr4 自动生成具体语法树 CST

通过 antlr4 工具，可以根据 g4 中定义的语法，自动生成一个词法分析器和一个句法分析器（在 antlr/gen 目录下），从而根据输入字符串生成具体语法树。基本步骤如下：

a) CharStreams 接受输入字符串流

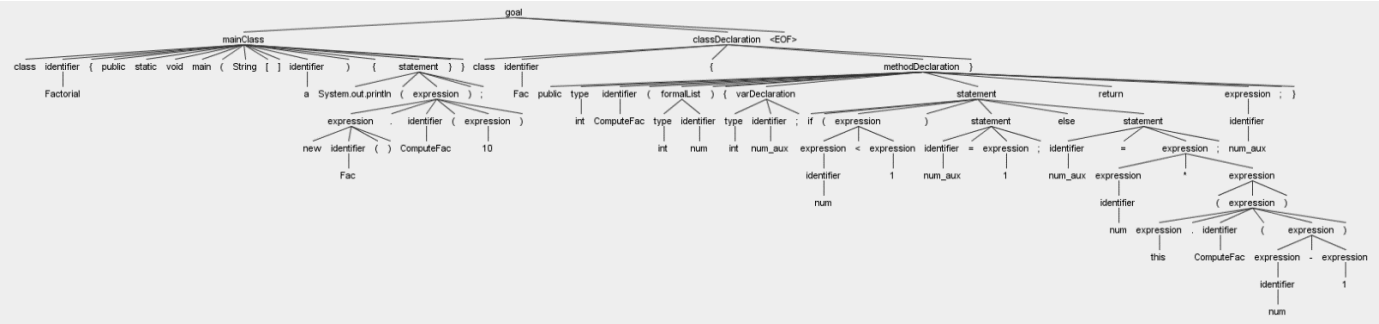
b) Lexer 将 charstream 转换成一个 token 流 CommonTokenStream

c) Parser 将 CommonTokenStream 根据句法规则转换成一棵语法树。语法树可以根据 antlr4 提供的接口 TreeView 可视化。

核心代码如下：

```
68              // create a lexer that feeds off of input CharStream
69              MyminiJavaLexer lexer = new MyminiJavaLexer(CharStreams.fromFileName(filename));
70              lexer.removeErrorListeners();
71              lexer.addErrorListener(new MyErrorListener.UnderlineLexerListener());
72
73              // create a buffer of tokens pulled from the lexer
74              CommonTokenStream tokens = new CommonTokenStream(lexer);
75
76              // create a parser that feeds off the tokens buffer
77              MyminiJavaParser parser = new MyminiJavaParser(tokens);
78              parser.removeErrorListeners(); // remove ConsoleErrorListener
79              parser.addErrorListener(new MyErrorListener.UnderlineParserListener());
80              parser.getInterpreter().setPredictionMode(PredictionMode.LL_EXACT_AMBIG_DETECTION);
81              tree = parser.goal();
```

以 examples/Factorial.java 为例，得到的具体语法树如下：



（2）重新遍历生成抽象语法树 AST

观察上述具体语法树，可以看到包括许多“{”“System.out.println”等常数字符串，这些字符串和程序执行的没有太大关系，而是和具体语言相关。由此需要重新遍历 CST，取出必要部分生成 AST。

使用 Antlr4 提供的 Visitor 接口，可以很方便的遍历 CST。为了生成 AST，并且保留树结构以便之后的语义检查使用，重新定义了一个树结构 ASTTree.class，里面定义了 AST 的结点，以及结点之间的关系。这一部分抽象语法主要根据 miniJava Project 的 Abstract Syntax 中 PrettyPrintVisitor.java 的定义。

在 Visitor 类中，每个 Parser rule 都对应着一个 visitor 函数。主函数通过以 CST 为参数调用 visitor，根据 CST 匹配的规则调用该规则对应的 visit 函数。每个 visit 函数以一个 ParserRuleContext 类型的变量 ctx 为参数，ctx 中包括了匹配该规则时刻的周围环境。可以通过在 g4 文件中定义的 Non-terminal 标识符访问该规则的右侧组成部分，同时可以通过 ctx 提供的函数定位此次匹配所在源文件的位置，从而辅助错误定位。

因此，在遍历 CST 时候，只需要从最开始的规则，遍历其右侧 Non-terminal 结点，每遍历一个结点，生成一个对应的 AST 结点，同时再递归调用 visitor 函数，最终生成一棵树。为了在 AST 上进行的语义检查能够定位错误，在建立 AST 结点的同时，需要传递该结点所在位置。

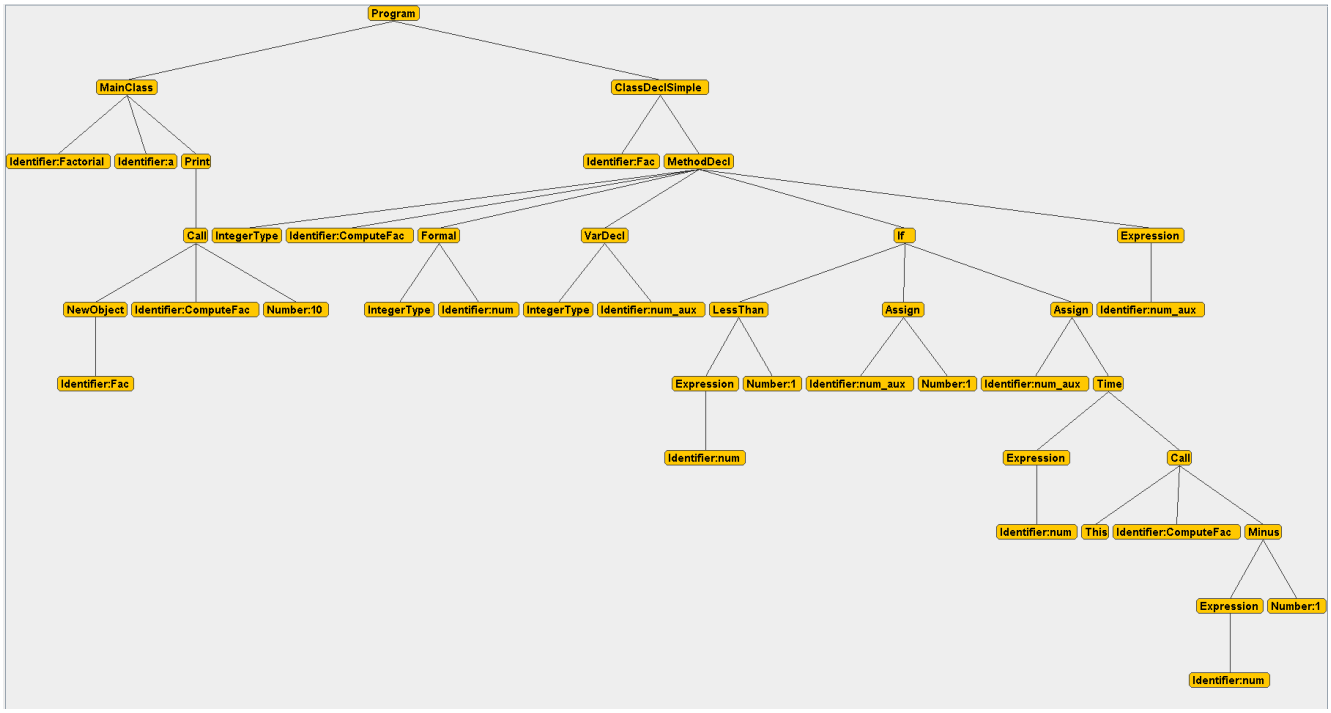
主函数的核心代码如下：

```
        // begin parsing at goal rule
        ASTtree.ASTtreeNode root = null;
        if(tree != null && !MyErrorListener.IsError()) {
            MyminiJavaASTVisitor ASTvisitor = new MyminiJavaASTVisitor();
            root = ASTvisitor.visit(tree);
            LISPtoTreeView.ShowLISPTree(root.printNode());
        }
        else{
            System.err.println("There are syntax errors and the AST tree can't be created!");
            return;
        }
```

其中 ASTvisitor.visit(tree)对 CST 进行遍历，并且返回 AST 的根节点。

root.PrintNode()通过在 AST 树上的递归输出，可以得到 LISP 格式的树表示。通过开源代码库 abego treelayout 的树可视化代码，将 LISP 先转化成标准树格式，再通过 swing 可视化出来。

以 examples/Factorial.java 为例，得到的抽象语法树如下：



3.3 遇到的问题及解决思路

（1）The following set of rule are manually made left-recursive：在 g4 文件生成的时候，提示左递归语法错误。

在 Antlr4 中，对于左递归的文法有一定的容忍性，允许直接的左递归，但是不允许间接的左递归。例如：

```
statement | expression;
expression | expression op expression
           | INTEGER_LITERAL
```

是合理的语法，尽管有 expression | expression op expression 这样的自身左递归。但是像下面的间接的左递归：

```
statement | expression;
expression | expression op expression
           | statement
```

| INTEGER_LITERAL

是会报错的。在这种情况下，往往需要利用消除左递归的算法对文法规则进行拆分，重写文法规则。

此外，在词法规则中是绝对不允许出现任何形式的左递归的。

在实验中，因为 lexer rule 和 parser rule 可以写在一个 g4 文件中，因此需要注意两种规则的区分，lexer rule 一定要以大写字母开头，parser rule 一定要以小写字母开头。在一开始的 g4 文件中，因为大小写混用导致 parser rule 被误认为 lexer rule，从而导致本身的左递归也会报错。

（2）单行注释和多行注释

miniJava 支持两种格式的注释，一种是“//”开头的单行注释，直到最后“\n”或者“\r”结尾；一种是“/**/”的多行注释，允许嵌套模式。

因为“\n”和“\r”都属于空白符，在词法分析阶段就会被 skip 规则丢弃，因此为了确保能够正确匹配单行注释，单行注释规则必须在词法分析阶段进行。利用内置的 lexer rule 的正则匹配符号：

~ NEGATES CHARACTERS

. MATCHES ANY CHARACTER IN THE RANGE 0x0000 ... 0xFFFF

因此得到单行和多行注释：

```
LineComment    :   '//' ~('\r' | '\n')* ->  channel(HIDDEN)  ;
MultiLineComment    :   '/*' (MultiLineComment|.)*? '*/'  -> channel(HIDDEN)    ;
```

对单行注释进行分析，含义是以“//”开头，后面跟 0 个或者多个非‘\r’和‘\n’字符。

对多行注释进行分析，含义是以“/*”开头，“*/”结尾，中间是 0 个或者多个任意字符，或者重复嵌套的多行注释。在这种情况下，多余的“*/”将不被匹配。

因为在分析代码过程中，注释往往被忽略，因此需要将其放在 Hidden channel 中。如果日后需要取出，则可以通过 getHiddenTokensToLeft 函数取出。

（3）ErrorListener 添加后依然是默认输出格式，不能区分 Lexer error 和 Parser Error。

ErrorListener 在使用 addErrorListener 之前，需要使用 removeErrorListeners 将原来默认的处理机制关掉，否则仍然使用的是 ConsoleErrorListener。Lexer 和 Parser 分别绑定不同的 ErrorListener，对于 lexer 和 parser 需要分别处理解绑和捆绑。

四、错误处理与修复

4.1 错误修复原理

4.1.1 词法和句法检查

Lexer 和 Parser 分别绑定一个 ErrorListener，由 Listener 监听并且处理在分析过程中的错误，输出“Line 1:10 error msg”格式的错误提示。Anltr4 的默认错误处理是 ConsoleErrorListener，可以通过继承 BaseErrorListener 类进行自己的错误处理。

在文法和句法处理上，MyErrorListener 为 Lexer 和 Parser 分别定制了处理函数，从而区分两种不同的错误。词法错误提示以“Lexical Error”开头，文法错误以“Syntax Error”开头，并且对两种错误进行分别计数。

Lexical Error 只输出错误所在行数和位置，以及错误修复提示。

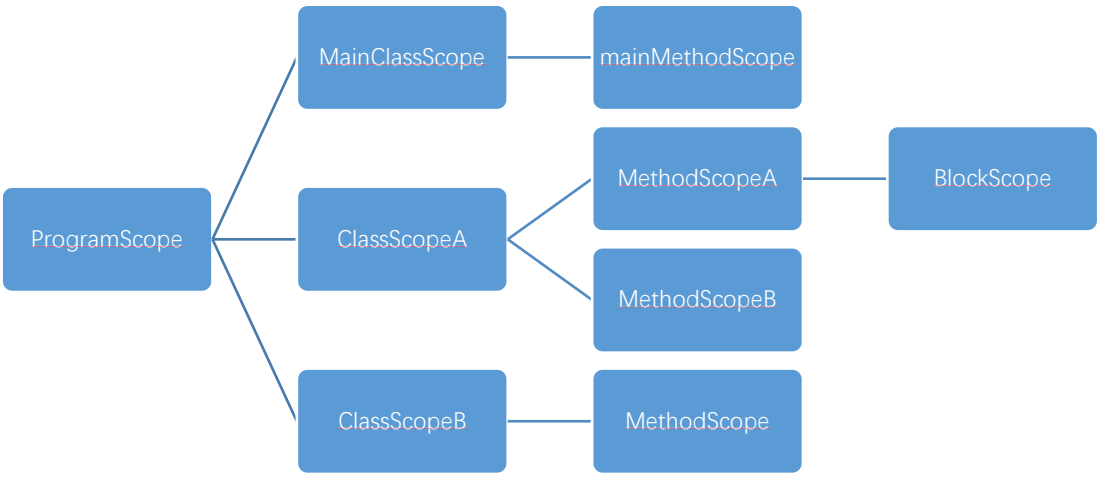
Syntax Error 给出错误所有的行数、位置以及错误修复提示之外，还参考《The Definitive ANTLR 4 Reference, 2nd Edition》中的示例给出出错所在的 Parser rule 层次，以及根据行数和位置给出的源代码位置提示。

4.1.2 语义检查

语义检查主要包括两部分，一部分是符号表 Symbol Table 的建立，一部分是类型检查 Type Check。

(1)建立符号表

因为变量在不同的作用范围内可以有不同的定义，因此符号表的建立和作用域紧密相关，而作用域是一个嵌套的结构，因此符号表也需要是嵌套的，这里使用一个树结构来存储符号表，大致的结构如下所示：



变量在使用时根据当前所在的作用域向上查找，直到找到或者最顶部作用域。在生成符号表时，可以检查是否重复定义变量。

以 examples/factorial.java 为例可以得到右图所示的符号表，每一张表第一行是作用域名，其下是符号表具体的条目。Kind 表示种类，有形参 arg，变量 var，类 class。Type 表示类型，有 IntegerType，IntArrayType，BooleanType 和自定义的类。对于函数来说，kind 是返回值的类型，type 是 func 类型，pos 是形参在函数定义中的位置，为了在调用函数的时候检查形参和实参对应。

```
class Factorial{
    public static void main(String[] a){
    System.out.println(new Fac().ComputeFac( num: 10));
    }
}

class Fac {

    public int ComputeFac(int num){
    int num_aux ;
    if (num < 1)
        num_aux = 1 ;
    else
        num_aux = num * (this.ComputeFac( num: num-1)) ;
    return num_aux ;
    }
```

```
==========The Symbol Table Hierarchy==========
mainScope
Factorial   kind:class  type:class  pos:-1
Fac kind:class  type:class  pos:-1

Factorial
main    kind:void   type:func   pos:-1

main
a   kind:arg    type:String[]   pos:-1

Fac
ComputeFac  kind:IntegerType    type:func   pos:-1

ComputeFac
num kind:arg    type:IntegerType    pos:0
num_aux kind:var    type:IntegerType    pos:-1
```

(2)类型检查

根据 miniJava 的类型定义，常见的类型检查有：

1) 变量和类中方法使用前定义

2) 二元操作左右类型：如四则运算需要都是 IntegerType，&&需要左右都是 BooleanType。

3) 赋值语句左右类型相等

4) 函数调用形参和实参数量和类型对应，返回值类型正确

5) If 和 while 的条件需要是 BooleanType 类型

通过对 AST 中每一种类型结点的检查，来递归实现语义检查。首先遍历一遍 AST，为生成符号表，再遍历依次，根据符号表检查类型是否正确。

4.2 错误修复结果

（1）词法和文法修复结果

以下图的错误代码为例。在第 11 行的变量名 b%不符合 identifier 的词法规则（identifier 只允许大小写字母和数字以及下划线），第 12 行的赋值语句缺了右侧表达式：

```
class Fmain{
    public static void main(String[] a){
    System.out.println(new F().func( num: 10));
    }
}

class F {

    public int func(int num){
        int a;
        int b%;
        a = ;
        return a;
    }

}
```

相应的错误提示为：

```
[1]Lexical Error: Line 11:13 token recognition error at: '%'
[2]Syntax Error: Line 12:12 mismatched input ';' expecting {'(', 'true', 'false', 'this', 'new', '!', IDENTIFIER, INTEGER_LITERAL}
    rule stack: [goal, classDeclaration, methodDeclaration, statement, expression]
          a = ;
              ^

There are 1 lexer error and 1 parser error.
There are syntax errors and the AST tree can't be created!
```

（2）语义修复结果

重复定义：examples/error/errorduplicate.java

```
class F {

    public int func(int num){
        int a;
        Fmain a;
        return a;
    }

}
```

```
[1]Semantic Error: Line 11:14 Duplicate var definition:a
[2]Semantic Error: Line 11:14 Type Error:a
    Require: Fmain, Get: IntegerType

There are 2 semantic error
```

未定义符号：examples/error/errordef.java

```
class F {

    public int func(int num){
        int a;
        b = num;
        return a;
    }

}
```

```
[1]Semantic Error: Line 11:8 Undefined Identifier:b
[2]Semantic Error: Line 11:8 Undefined Identifier:b
[3]Semantic Error: Line 11:12 Type Error in Assign statement
    Require: null, Get: IntegerType
There are 3 semantic error
```

F 中的 get 方法未定义，导致 func 的返回值不正确：examples/error/errorfun.java

```
class F {
    public int func(int num){
        int a;
        F b;
        b = new F();
        a = num;
        return b.get(10);
    }
}
```

```
[1]Semantic Error: Line 14:17 Undefined Identifier:get
[2]Semantic Error: Line 14:17 Undefined Identifier:get
[3]Semantic Error: Line 14:15 Type Error in Call Object
    Require: func, Get: null
[4]Semantic Error: Line 14:17 Undefined Identifier:get
[5]Semantic Error: Line 14:17 Undefined Identifier:get
[6]Semantic Error: Line 14:15 Type Error in Call Object
    Require: func, Get: null
[7]Semantic Error: Line 9:15 Return value error::func
    Require: IntegerType, Get: null

There are 7 semantic error
```

F 的方法 func 的调用参数类型和方法不对：examples/error/errorarg.java

```
class F {
    public int func(int num){
        int a;
        boolean t;
        F b;
        b = new F();
        a = b.func(t);
        a = b.func(num, num);
        return num;
    }
}
```

```
[1]Semantic Error: Line 14:19 Args Type Error in Call Object
    Require: IntegerType, Get: BooleanType
[2]Semantic Error: Line 15:19 Args Number Error in Call Object
    Require: 1, Get: 2
There are 2 semantic error
```

二元操作类型不符合，条件语句不为 true，赋值语句左右不一致：examples/error/errorop.java

```
class F {
    public int func(int num){
        int a;
        boolean b;
        a = 1;
        b = true;
        if(a + b)
            a = true;
        else
            a = false;
        return a;
    }
}
```

```
[1]Semantic Error: Line 13:15 Type Error in Binary Expression
    Require: IntegerType, Get: BooleanType
[2]Semantic Error: Line 13:15 Type Error in Binary Expression
    Require: IntegerType, Get: BooleanType
[3]Semantic Error: Line 13:11 Type Error in If statement
    Require: BooleanType, Get: IntegerType
[4]Semantic Error: Line 14:16 Type Error in Assign statement
    Require: IntegerType, Get: BooleanType
[5]Semantic Error: Line 16:16 Type Error in Assign statement
    Require: IntegerType, Get: BooleanType
There are 5 semantic error
```

五、 项目感想

项目主要为 miniJava 语法生成了一个编译器前端，并且带有一定的错误提示和修复能力。Antlr4 工具十分容易上手，生成的代码结构也很清晰，前期工作比较简单。

但是语义分析部分，需要自行构造符号表的结构层次，根据作用域建立符号表。因为 miniJava 和 Java 特性一样，以类来组织，因此符号表必须按照函数式风格（function style）而不是命令式风格（imperative syle）组织，即必须保存每一个作用域的变量，而不是毁坏之前的作用域，用到再恢复。静态的类型检查包括许多种不同情况。为了使得语义分析中的错误提示能够定位，在建立 AST 的时候必须传递结点位置。这一部分逻辑比较复杂和繁琐。

通过该项目初步体会了编译的前端工作，将源语言翻译成 AST 这种中间代码表示。通过 Antlr4 工具的使用，也加深了对于词法分析，语法分析，语义分析的理解。