编译Project: Compiler for miniJava

14307130360 王丹青

1. 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。

1. 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。

1. 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：

1. 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
2. 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。
3. 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。

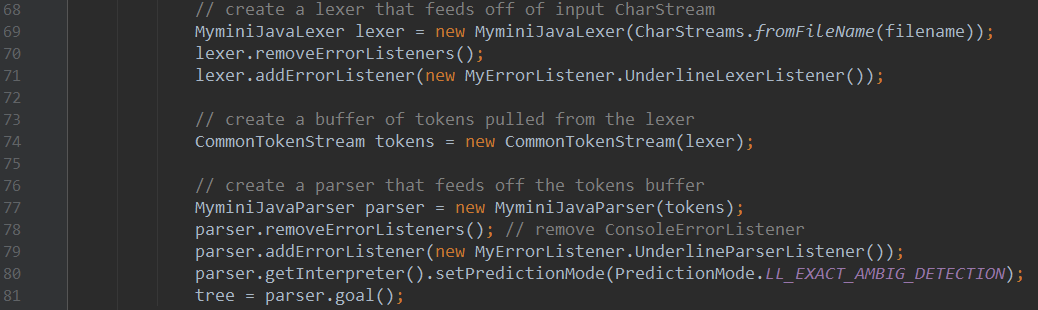
3.2.2 生成抽象语法树

（1）Antlr4自动生成具体语法树CST

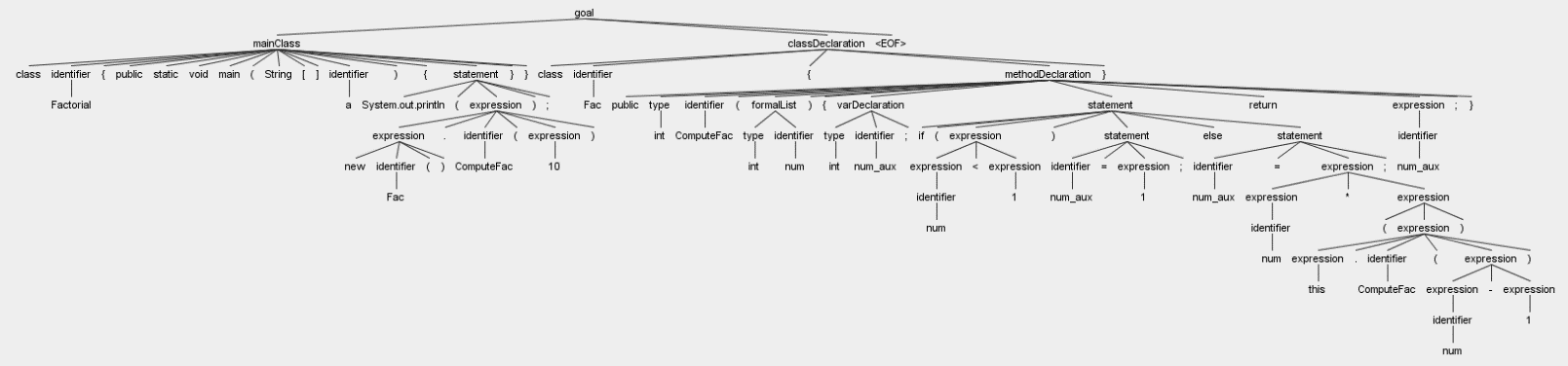
通过antlr4工具，可以根据g4中定义的语法，自动生成一个词法分析器和一个句法分析器（在antlr/gen目录下），从而根据输入字符串生成具体语法树。基本步骤如下：

1. CharStreams接受输入字符串流
2. Lexer将charstream转换成一个token流CommonTokenStream
3. 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：



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

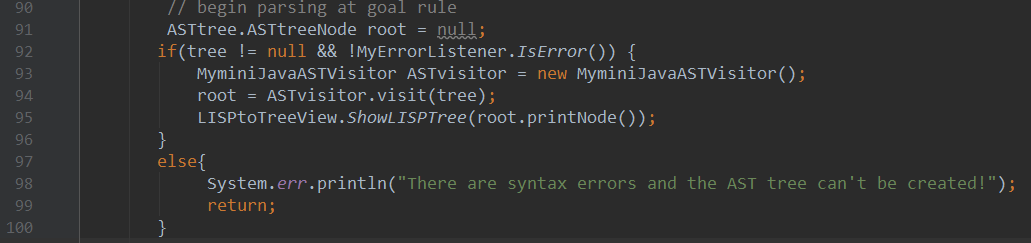
观察上述具体语法树，可以看到包括许多“{”“System.out.println”等常数字符串，这些字符串和程序执行的没有太大关系，而是和具体语言相关。由此需要重新遍历CST，取出必要部分生成AST。

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](http://www.cambridge.org/us/features/052182060X/)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。

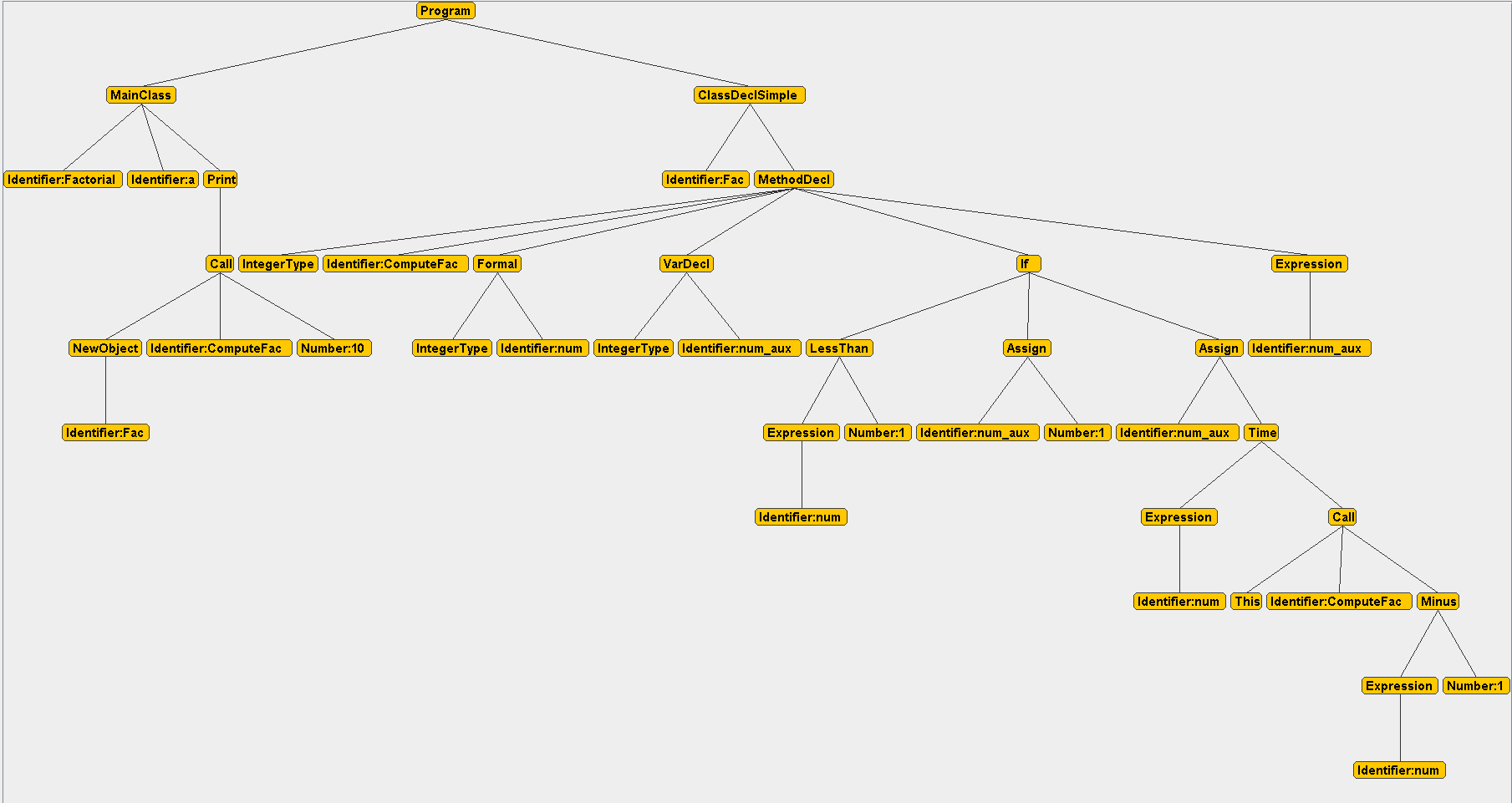
主函数的核心代码如下：



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](http://treelayout.sourceforge.net/)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：



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。

1. 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

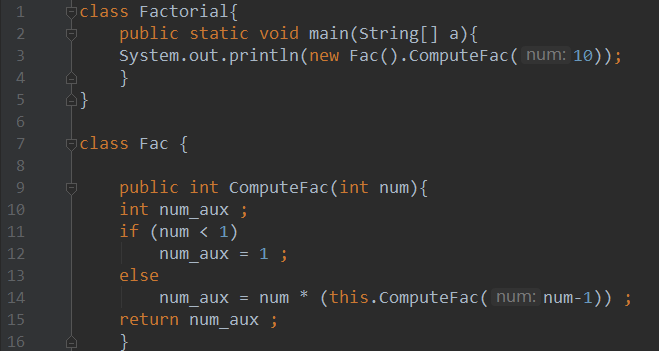
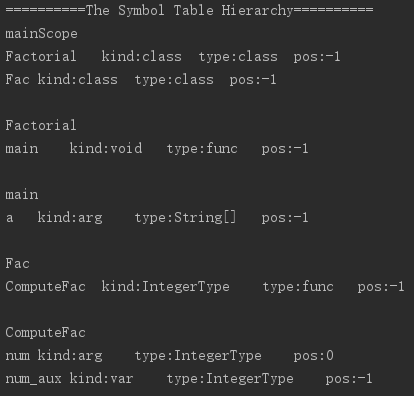
语义检查主要包括两部分，一部分是符号表Symbol Table的建立，一部分是类型检查Type Check。

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。

1. 类型检查

根据miniJava的类型定义，常见的类型检查有：

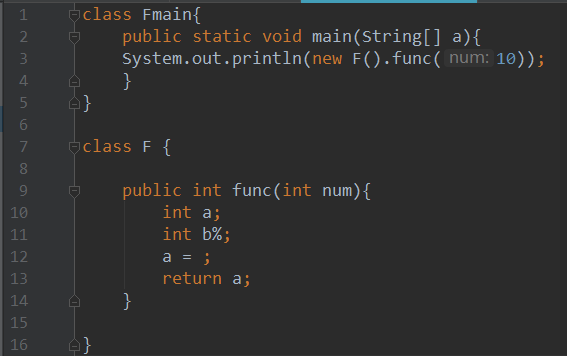
1. 变量和类中方法使用前定义
2. 二元操作左右类型：如四则运算需要都是IntegerType，&&需要左右都是BooleanType。
3. 赋值语句左右类型相等
4. 函数调用形参和实参数量和类型对应，返回值类型正确
5. If和while的条件需要是BooleanType类型

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。

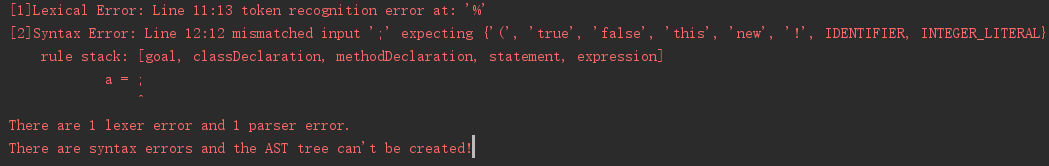
4.2 错误修复结果

（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：

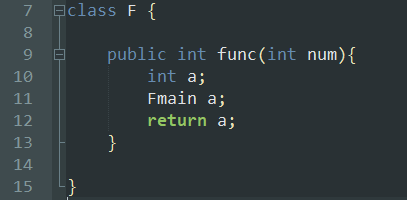
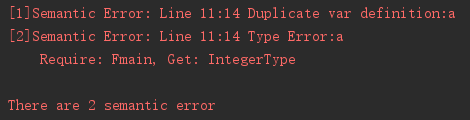


相应的错误提示为：

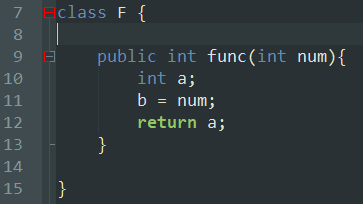
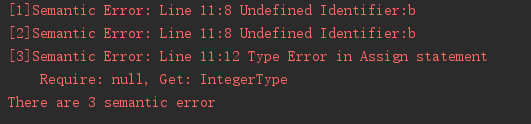


（2）语义修复结果

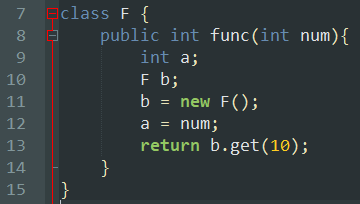
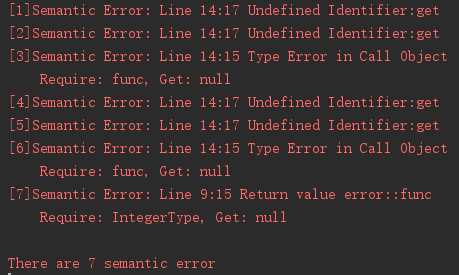
重复定义：examples/error/errorduplicate.java

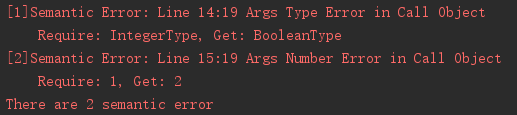
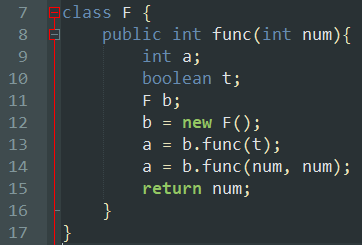
未定义符号：examples/error/errordef.java

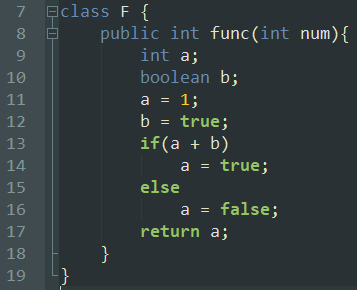
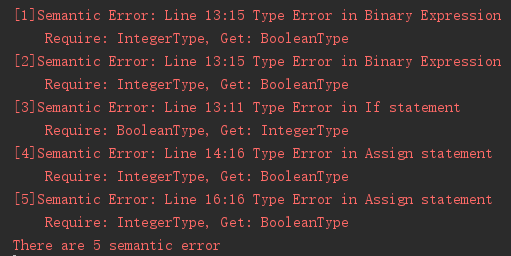
F中的get方法未定义，导致func的返回值不正确：examples/error/errorfun.java

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



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

1. 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。