编译 Project 报告

段宇　14307130262

汤佳欣 14307130359

1. 工具选择

1.1 ANTLR（Another Tool for Language Recognition）

一种自动化工具，可以通过定义的语言规则，为用户生成相应的词法和语法分析器。

词法分析器（Lexer）是分析字符流将它们处理成离散的字符组并标注类型，包括关键字、标识符、操作符等供语法分析器使用。语法分析器（Parser）只需要关注单词的类型，将词法分析器的结果组织起来，转化成为目标语言语法定义所允许的序列。

ANTLR 将上述两者结合起来，允许用户定义识别字符流的词法规则和用于解释 Token 的语法规则（遵循 ANTLR 的元语言语法），根据用户提供的语法文件自动生成相应的词法/语法分析器，并建立了一个语法分析树的数据结构。ANTLR 采用的语法分析方法是 LL(K)。

因本 project 选用该工具，具体使用见本报告第二部分。

1.2 Lex/Yacc

Lex 是一个词法分析自动化工具，lex 文件是一个包含一系列词法规则的文本文件，按照这些规则自动生成 C 函数 yylex()，这个函数把字符串作为输入，按照定义好的规则分析字符串中的字符，找到符合规则的字符序列后，执行在规则中定义好的动作。

Yacc 是适合上下文无关文法，采用 LALR(1)方法的语法分析工具，对于移进-归约冲突，Yacc 的设计是，用移进来解决冲突，除非有操作符优先级声明（使用%left 和%right 指定）的指令。

1.3 Flex/Bison

Flex 是有 Vern Paxon 实现的 Lex，Bison 则是 GNU 版本的 YACC。

1.4 Jlex/CUP

Jlex 是用 Java 重写的 Lex，用来自动产生 Java 源代码的词法分析程序，CUP 则自动生成与之对应的 Java 源代码的语法分析器。

1.5 JavaCC

JavaCC 是一个用 Java 写的 Java 词法、语法分析器的生成器。Javacc 采用的是 LL 语法分析方法，没有回溯功能，因此程序员需要解决冲突问题：通过经典的修改文法的方法解决左递归问题，当碰到冲突时，修改语法使之成为 LL(1)语法或在 Options 块中添加 LOOKAHEAD=k

来使 javacc 可以分析 LL(K)语法。

注：

• LL 语法分析器 LL(k)

递归下降分析，从起始符号开始，比较输入中的终结符和文法规则，判断符合哪个产生式规则，k 表示允许一次查看的终结符的数量。递归下降分析需要解决左递归、公因子等问题。

• LR 语法分析器 LR(k)

从左至右分析、最右推导、超前查看 k 个单词，分析器有一个栈和一个输入，输入中的前 k 个单词为超前查看的单词。根据栈的内容和超前查看的单词，分析器执行移进和归约两种动作：移进是将第一个输入的单词压入栈顶；归约是从栈顶依次弹出产生式右侧的单词，然后将左侧的非终结符压入栈。

LALR：LR 分析表有很多状态，非常大。通过合并除超前查看符号集外都相同的两个状态，可得到一个较小的表，由此得到的分析器即 LALR 分析器。

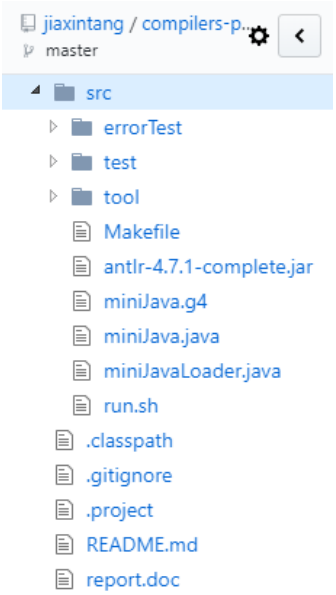
• 选用 ANTLR 的原因：

由于 LR 语法分析器是在得到了更多的输入的前提下做出判断的，所以一般来说，LR 分析器都要比 LL(K)分析器的解析能力更强。但相反地，由于需要更多的输入，导致了语法分析器确定性的解析策略，因此对于一些写法很不好的代码，会解析失败。

ANTLR4 通过透明的将左递归替换为一个判定循环而允许在 LL 语法描述中使用左递归。虽然它能够只能处理直接的左递归，不能处理间接的左递归，但间接的左递归在一般的语法描述中很少见。

2. 具体实现

2.1 代码结构



src/ :项目代码

src/tool/ :开发小工具

src/test/ :无编译错误的测试代码

src/errorTest/ :有编译错误的测试代码

src/miniJava.g4 :ANTLR4 文法文件

src/miniJava.java :编译器前端主程序

src/miniJavaLoader.java :项目核心代码文件，重载 ParseListener 生成抽象语法树、进行语义检查。

2.2 核心代码工作原理

2.2.1 词法、语法规则文件撰写

ANTLR 的词法规则和语法规则统一写在 g4 文件中，词法名以大写字母开头，语法名以小写字母开头。参考 http://www.cambridge.org/us/features/052182060X/grammar.html 给出的 MiniJava 语法规则，由于 ANTLR 可以处理左递归，所以可以保留类似如下语法规则：

```
expression
    :expression DOT ID LPR ( expression ( COMMA expression )* )? RPR
    |expression DOT 'length'
    |expression SLP expression SRP
    |expression EXP<assoc=right> expression
```

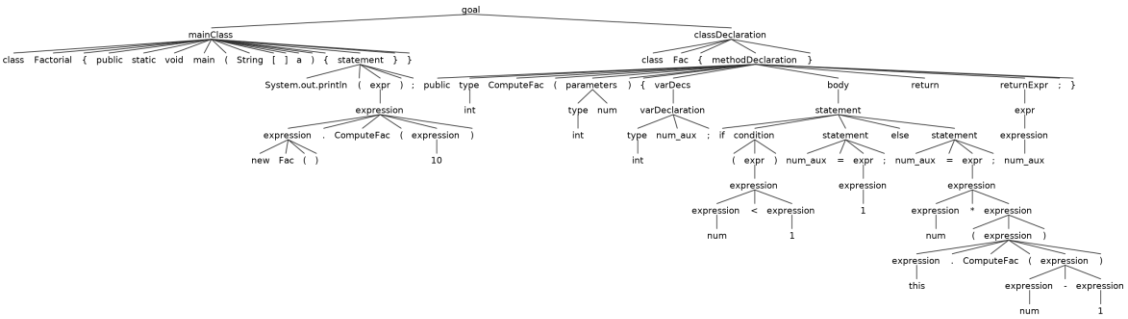
在语法规则后添加标签，ANTLR 会根据标签生成与规则相关的解析树的事件监听函数，方便语义分析时在各节点的操作。

```
expression
    :expression DOT ID LPR ( expression ( COMMA expression )* )? RPR   #method
    |expression DOT 'length'                                           #length
    |expression SLP expression SRP                                     #access
    |expression EXP<assoc=right> expression                            #exp
    |expression MUL expression                                         #mul
    |expression (OP=ADD | OP=SUB) expression                           #addSub
    |expression EQ expression                                          #EQ
    |expression LT expression                                          #LT
    |expression AND expression                                         #and
    |INT                                                              #int
    |FLOAT                                                            #float
    |BOOLEAN                                                          #bool
    |STRING                                                           #string
    |ID                                                               #id
    |THIS                                                             #this
    |'new' 'int' SLP expression SRP                                   #newInt
```

2.2.2 语法分析树的生成

使用 ANTLR4 工具生成的语法分析树。

Make rig TESTFILE=”$(YOURFILE)”



2.2.3 抽象语法树的生成

选择使用重载 Listener 进行分析。Listener 使用深度优先搜索遍历语法分析树，在每个节点退出时，根据深度优先搜索的特性，该节点下的所有子节点已经结束访问。创建一个新节点，

根据当前节点的类型，把某几个子节点作为新节点的儿子。此时生成了抽象语法树的结构，还需要对每个节点的显示文本进行设置。ANTLR 中有对语法树进行显示的函数，显示时会调用每个节点的 getRuleIndex()函数获取文本列表的下表。所以我们对每个类型的节点进行重载，设置 getRuleIndex()的返回值为我们自定义的下标。具体实现如下：

生成抽象语法树(把需要的子节点添加成新节点的儿子)：

```
@Override public void exitGoal(miniJavaParser.GoalContext ctx) {
    ParserRuleContext node = new GoalContext2(ctx, 0);
    node.copyFrom(ctx);
    try{vast.get(ctx.mainClass()).getText();node.addChild(vast.get(ctx.mainClass()));}catch (NullPointerException e) {}
    for (miniJavaParser.ClassDeclarationContext i: ctx.classDeclaration())
        try{vast.get(i).getText();node.addChild(vast.get(i));}catch (NullPointerException e) {}
    node.invokingState = _GOAL;
    ast = node;
}
```

显示文本列表：

```
List<String> l = new ArrayList<>();
l.add("goal"); // 0
l.add("main"); // 1
l.add("classDec"); // 2
l.add("Dec"); // 3
l.add("methodDec"); // 4
l.add("type"); // 5
l.add("state"); // 6
l.add("expr"); // 7
l.add("body"); // 8
l.add("if"); // 9
l.add("while"); // 10
l.add("print"); // 11
l.add(":="); // 12
l.add("ID[expr]=expr"); // 13
l.add("expr.func(expr*)"); // 14
l.add("expr.length"); // 15
l.add("expr[expr]"); // 16
l.add("^"); // 17
l.add("*"); // 18
l.add("+/-"); // 19
l.add("=="); // 20
l.add("<"); // 21
l.add("&&"); // 22
l.add("int"); // 23
l.add("bool"); // 24
l.add("ID"); // 25
l.add("this"); // 26
l.add("new int[expr]"); // 27
l.add("new ID()"); // 28
l.add("!"); // 29
l.add("()"); // 30
l.add("return"); //31
l.add("param"); //32
l.add("vars"); //33
l.add("body"); //34
l.add("condition"); //35
l.add("float"); //36
l.add("expr(WRONG!)"); //37
l.add("string"); //37
```

文本下标设置：

```
public static final int
    _GOAL=0, _MAINCLASS=1, _CLASSDECLARATION=2, _VARDECLARATION=3,
    _METHODDECLARATION=4, _TYPE=5, _STATEMENT=6, _EXPRESSION=7,
    _BLOCK=8, _SELECT=9, _WHILE=10, _OUTPUT=11, _ASSIGN=12,
    _ARRAYASSIGN=13, _METHOD=14, _LENGTH=15, _ACCESS=16,
    _EXP=17, _MUL=18, _ADDSUB=19, _EQ=20, _LT=21,
    _AND=22, _INT=23, _BOOL=24, _ID=25, _THIS=26,
    _NEWINT=27, _NEWID=28, _NOT=29, _PAREN=30,
    _RETURNEXPR=31, _PARAMETERS=32, _VARDECS=33,
    _BODY=34, _CONDITION=35, _FLOAT=36,
    _EXPR=37, _STRING=38;
```

树节点继承与函数重载：

```
public static class GoalContext2 extends miniJavaParser.GoalContext {
    @Override public int getRuleIndex() {return _GOAL;}
    public GoalContext2(ParserRuleContext ctx, int invokingState) {super(ctx, invokingState);}
}
```

样例：对于如下 MiniJava 代码：

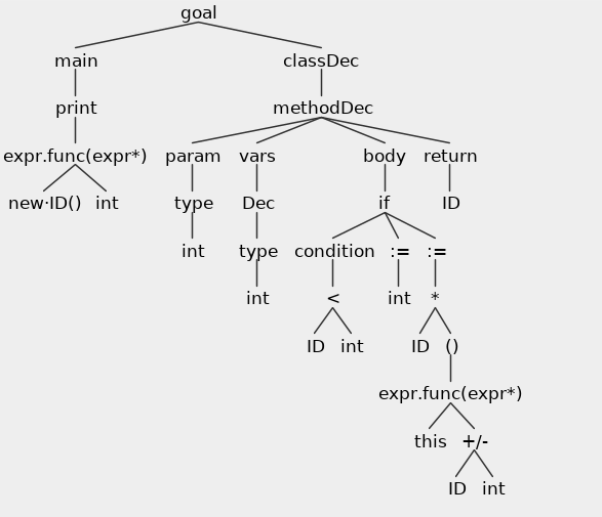
```
class Factorial{
    public static void main(String[] a){
        System.out.println(new Fac().ComputeFac(10));
    }
}

class Fac {

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

}
```

生成的抽象语法树如下：



2.2.4 语义分析

在 ANTLR4 中，有两种遍历语法分析树的方式，visitor 和 listener：visitor 是语法分析树节点的迭代器，可以主动访问节点，可以通过重载每个节点的 visitor，定义访问时要进行的操作；listener 是语法分析树节点的监听器，在进入和退出时每个节点时，会触发回调函数，可以通过重载每个节点的 listener 定义回调要进行的操作。具体类型检查的步骤见 2.3.3 语义错误处理与修复。

2.3 错误处理与修复

2.3.1 词法错误处理与修复

```
INT : [0-9]+;
WRONG_INT : [0-9][a-zA-Z0-9_]+;
UNAVALIABLE_CHAR : [$%@#|/'`~]+;

identifier
    : ID
    | UNAVALIABLE_CHAR {notifyErrorListeners("illegal character");}
    ;
number
    : INT
    | WRONG_INT {notifyErrorListeners("illegal number");}
    ;
```

主动构造错误的词法类型，将其和 ID 及 INT 一同构造到语法中，通过语法分析的提示显示词法错误。

样例：

```
1 class WordsError{
2     public static void main(String[] a) {
3         System.out.println(1a123+1_);
4     }
5 }
```

```
→  src git:(master) ✗ ./run.sh errorTest/wordsError.java
line 3:26 Wrong number
line 3:29 Wrong number
```

2.3.2 语法错误处理与修复

```
expr
    :expression RPR { notifyErrorListeners("Too many parentheses"); }
    |LPR expression { notifyErrorListeners("Missing closing ')'"); }
    |expression;
```

枚举可能出现的出现类型，在语法中构造常见的语法错误，当匹配到对应的错误语法时，使用”notifyErrorListeners”函数进行错误提示。并且不会影响后续的匹配（实现了错误修复）。

样例：

```
1 class SyntaxError{
2     public static void main(String[] a) {
3         System.out.println("");
4     }
5 }
6 class F{
7     public int s(){
8         int a;
9         a = ((1+2);
10    }
11 }
```

```
→  src git:(master) ✗ ./run.sh errorTest/SyntaxError.java
line 9:12 Missing closing ')'
line 11:0 missing 'return' expression
```

2.3.3 语义错误处理与修复

使用如下几个表来进行类型检查：

节点类型表、类名表、类型方法参数表、类变量表、方法变量表

语义分为以下几步：

1. 遍历类名，加入类名表

2. 分析继承关系，生成类的继承有向图

3. 使用拓扑排序，依次构造类变量表和类型方法参数表

4. 在遍历抽象语法树时，当进入方法声明节点时，先访问参数表，添加参数表中的变量到方法变量表，然后将变量声明部分的变量加入变量表；离开方法声明节点时，将方法变量表清空。

5. 表达式运算的每个节点，通过节点类型表获取其子表达式的类型，判断表达式是否符合运算要

求，将计算得到的类型存入节点类型表。

6. 类型声明节点，查询类名表是否存在类型，并将声明的变量及其类型加入方法变量表中。

7. ID 节点访问时，通过方法变量表、类变量表查询该 ID 是否存在，并将其类型存入节点类型表。

8. 当表达式出现错误时，给出错误提示并将其类型表示为“wrong”，当发现有类型为“wrong”时不再出现错误提示。

表达式类型检查(加减表达式)：

```
919
920         String typeL = values.get(ctx.expression(0));
921         String typeR = values.get(ctx.expression(1));
922         TerminalNode op = ctx.OP.getText().equals("+")?ctx.ADD():ctx.SUB();
923         if (typeL.equals("wrong") || typeR.equals("wrong")) {
924             values.put(ctx, "wrong");
925             return;
926         }
927         if ((typeL.equals("int") || typeL.equals("float")) && (typeR.equals("int") || typeR.equals("float"))) {
928             if (typeL.equals("float") || typeR.equals("float"))
929                 values.put(ctx, "float");
930             else
931                 values.put(ctx, "int");
932             if (!typeL.equals(typeR))
933                 warn(op, "Implicit conversion between 'int' and 'float'");
934         }
935         else {
936             err(op, "Invalid operands of type '" + typeL + "' and '" + typeR + "' to binary " + ctx.OP.getText());
937             values.put(ctx, "wrong");
938         }
939     }
```

如图，获取左右表达式的类型，判断子表达式是否错误。然后判断左右表达式是否符合要求，最后设置节点类型。

具体错误处理及截图如下：

2.3.3.1 类、方法、变量重定义

```
1 class Duplicate{
2     public static void main(String[] a){
3         System.out.println("");
4     }
5 }
6 class A {}
7 class A {}
8 class B {
9     int c;
10    int c;
11    public int f() {
12        return 1;
13    }
14    public float f() {
15        return 2.;
16    }
17 }
```

```
➜  src git:(master) ✗ ./run.sh errorTest/Duplicate.java
line 7:6 error : Class 'A' already exists
class A {}
      ^

line 14:14 error : Function name 'f' already declared
 public float f() {
              ^

line 10:5 error : redefinition of 'int c'
 int c;
     ^
```

(1) 重复定义同名的类：提示 error，类型已存在。

(2) 重复定义同名方法：提示 error，函数名已存在。

(3) 重复定义同名函数：提示 error，变量名重定义。

2.3.3.2 未定义错误

```
1 class Undefine{
2     public static void main(String[] a){
3         System.out.println(new A().f());
4     }
5 }
6 class A{
7     public int func() {
8         return a;
9     }
10 }
11 class B extends A{
12     int a;
13     public int f() {
14         return a;
15     }
16 }
17 class D extends C{}
```

```
→  src git:(master) ✗ ./run.sh errorTest/Undefine.java
line 14:16 error : Unknown class 'C' found when extends
class D extends C{}
                ^

line 3:29 error : class 'A' has no member named 'f'
  System.out.println(new A().f());
                             ^

line 8:9 error : 'a' was not declared in this scope
  return a;
         ^
```

(1) 主函数调用类 A 中未定义的函数 f：提示 error，类 A 没有成员’f’。（虽然 B 中定义了 f 函数，但不在类 A 中）

(2) 类 A 中的方法 func 使用了未定义的变量 a：提示 error，a 不在当前作用域。（B 中有定义 a，但不在 A.func()的作用域中）

(3) 类 D 继承类 C，但没有类的名字为 C：提示 error，未找到类名’C’。

2.3.3.3 循环继承错误

```
class LoopExtends{
    public static void main(String[] a){
        System.out.println("");
    }
}
class A extends B{}
class B extends C{}
class C extends A{}
```

```
→  src git:(master) ✗ ./run.sh errorTest/LoopExtends.java
line 8:16 error : Loop inheritance occurred when extends 'C'
class C extends A{}
                ^
```

(1) 类 A->类 B->类 C->类 A，出现了循环继承：提示 error，在继承时出现环。

2.3.3.4 类方法继承

```
class ExtendsMethods{
    public static void main(String[] a){
        System.out.println(new test().test());
    }
}
class A{
    public int f(){
        return 1;
    }
}
class B extends A{}
class C extends B{}
class D{}
class E extends D{
    public int f(){
        return 1;
    }
}
class test {
    public int test(){
        System.out.println(new A().f());
        System.out.println(new B().f());
        System.out.println(new C().f());
        System.out.println(new D().f());
        System.out.println(new E().f());
        return 1;
    }

}
```

```
→  src git:(master) ✗ ./run.sh errorTest/ExtendsMethods.java
line 24:29 error : class 'D' has no member named 'f'
  System.out.println(new D().f());
                             ^
```

(1) 类 A 中有函数 f，类 B 继承类 A，类 C 继承类 B，调用 new A().f()、new B().f()、new C().f() 不报错。

(2) 类 D 中没有 f 函数，调用 new D().f()：提示 error，找不到成员’f’。

(3) 类 E 继承自类 D，增加了函数 f，调用 new E().f()不报错。

2.3.3.5 错误恢复

即遇到错误，尝试修复，能继续编译下面的代码，而不是报错后结束编译。

实现方法：当表达式出现错误时，给出错误提示并将其类型表示为“wrong”，当发现有类型为“wrong”时不再出现错误提示。同时每句语法之间是独立分析的，某个表达式错误不会直接退出分析，所以可以实现错误修复。

```
class TypeCheck{
    public static void main(String[] a){
        System.out.println(new test().test("asd"));
    }
}
class test {
    int a;
    int a;
    int b;
    public int test(int c){
        int b;
        float d;
        boolean e;
        d = c + a * b;
        d = d + b;
        if (a)
            d = e;
        else if (d == e)
            e = (b==a);
        else
            e = b;
        return e;
    }
}
```

```
➜  src git:(master) ✗ ./run.sh errorTest/TypeCheck.java
line 3:32 error : function 'test' expect 'int'. but 'String' got
  System.out.println(new test().test("asd"));
                               ^^^^

line 8:5 error : redefinition of 'int a'
 int a;
     ^

line 14:4 warning : Implicit conversion between 'int' and 'float'
  d = c + a * b;
    ^

line 15:8 warning : Implicit conversion between 'int' and 'float'
  d = d + b;
        ^

line 16:5 warning : Implicit conversion from 'int' to 'boolean'
  if (a)
     ^

line 17:5 error : Invalid operands of type 'float' and 'boolean' to binary '='
   d = e;
     ^

line 18:13 error : Invalid operands of type 'float' and 'boolean' to binary ==
  else if (d == e)
             ^^

line 21:5 error : Invalid operands of type 'boolean' and 'int' to binary '='
   e = b;
     ^
```

2.4 额外功能说明

2.4.1 隐式类型转换警告

2.4.2 增加 String 类和 float 类

```
1 class Implicit{
2     public static void main(String[] a){
3         System.out.println(new A().func());
4     }
5 }
6 class A{
7     public int func() {
8         int a;
9         float b;
10        a = b;
11        return a;
12    }
13 }
```

```
➜  src git:(master) ✗ ./run.sh errorTest/Implicit.java
line 10:4 warning : Implicit conversion between 'int' and 'float'
  a = b;
    ^
```

3. 项目感想

该 project 主要实现了 miniJava 的编译器前端，运用 ANTLR 语法分析工具实现了词法和语法分析，并在此基础上实现了语义分析、词法语法和语义的错误信息处理和修复。通过这次 project，我们不仅对于编译器的词法、语法、语义分析每个步骤及分界都有了更深入清晰地理解，对这门课的整体框架也有了更全面地感知。

一开始我们对于 ANTLR 的使用不太了解，于是阅读了 ANTLR4 的使用文档以及 API 接口，在这个过程中，提高了文档的查阅能力。开发中使用 Java，也学到了 Java 的设计模式。

ANTLR 的监听器设计很奇怪，不能主动修改节点的 Index，这对我们构建抽象语法树造成了非常大的阻碍，语法分析树的 GUI 中，默认文本十分不准确，需要手动指定，但输出语法树 GUI 的函数需要根据节点的 Index 来对应文本列表下标。文档中没有涉及到如何进行修改，于是去阅读了 ANTLR4 的源代码，发现这是固定的，唯一的解决方案就是自行继承树节点并重载修改。最后我们将所有出现的标签都继承重载了一次，终于得到了正确合理的抽象语法树。