任务二会议纪要

本文档说明

本小组项目的学习以周期性会议的形式开展，每次时间不少于3小时，每周学习时间不少于9小时，因此可能会出现某次会议的末尾和下一次会议的开始都是类似学习内容的情况。故文档以项目的学习内容为架构，不以会议时间等为标题。但会在任务全部完成后，在本文档的开头汇总每次会议的纪要以总结体现小组认真的态度。而之后细节的工作记录也有会议时间的体现，细致记录了每次的会议和学习内容。

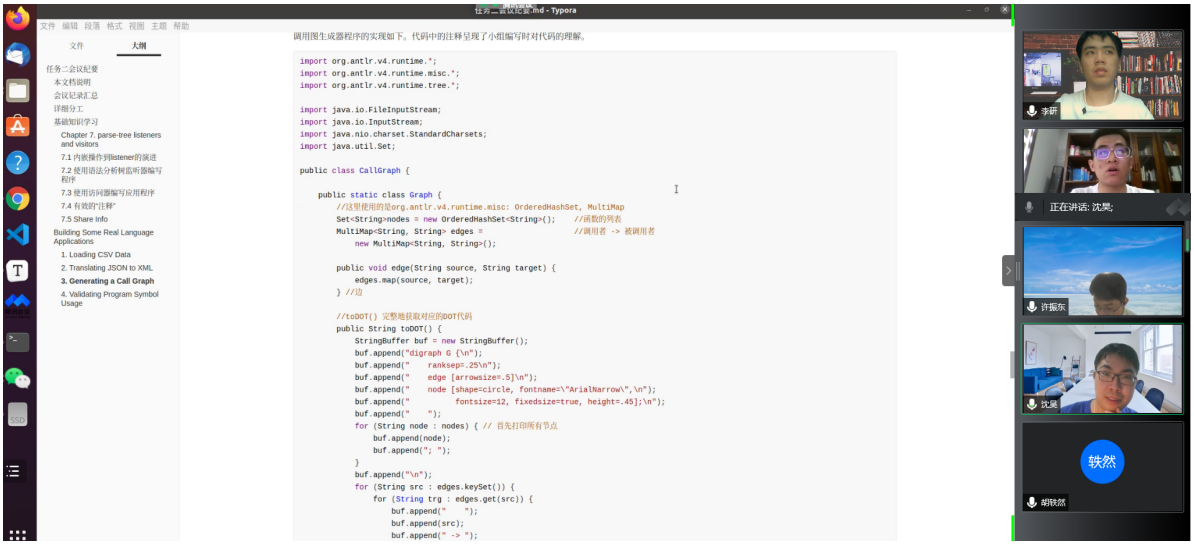
会议纪要记录人：胡轶然

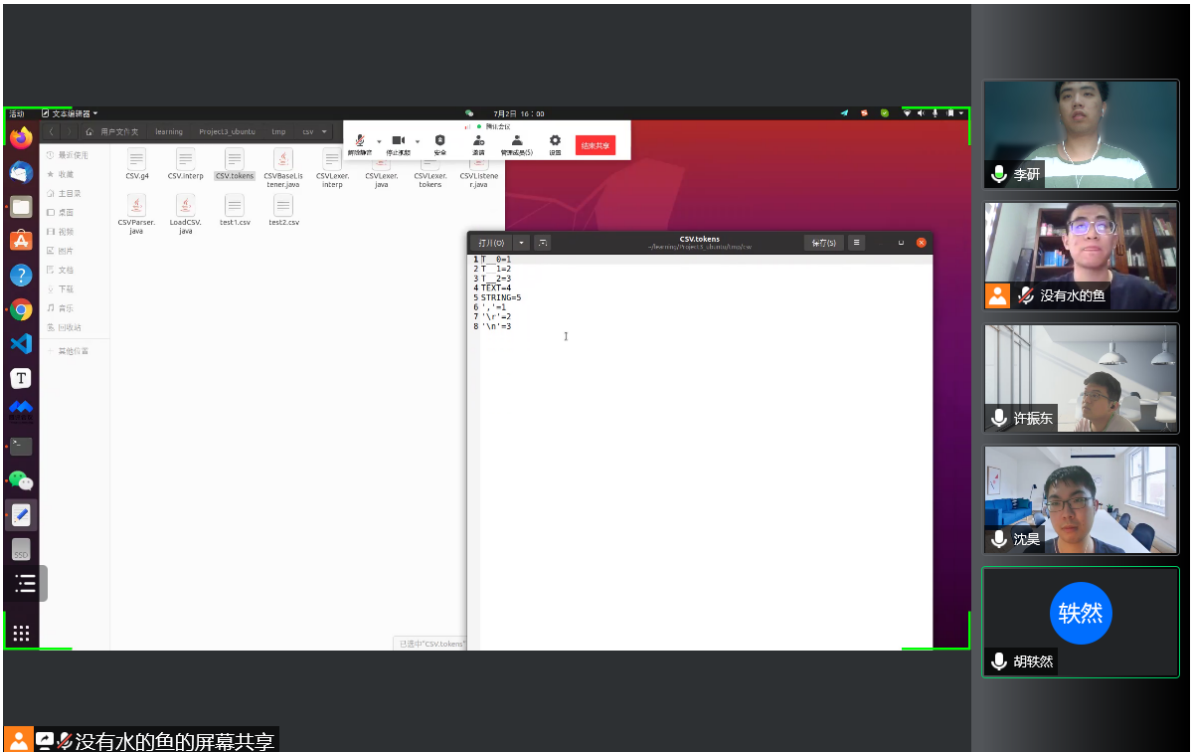
会议纪要审核人：沈昊

会议记录汇总

1. 任务二第1次会议：2022.3.18 19:00-22:30 项目制3的第4次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），开启任务二，精读参考资料第七章。细致学习了 listener和visitor的原理，并做了详细的记录，为进入第八章正式开始任务二打基础。
2. 任务二第2次会议：2022.3.20 19:00-22:30 项目制3的第5次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），小组继续任务二，实现8.1、8.2的任务，并跟进会议记录和设计文档。
3. 任务二第3次会议：2022.3.23 14:00-17:00 项目制3的第6次会议（参会人：胡轶然、沈昊、李研、胡书豪），小组继续任务二，实现任务8.3、8.4，完善检查会议纪要和设计文档，并开启任务三的摸索与第九章的学习。至此，任务二全部完成。

会议照片：





详细分工

胡轶然：任务二会议纪要的整理和书写；学习课本第七章listener和visitor的知识；完成第八章Building Some Real Language Applications 中**Validating Program Symbol Usage**的任务，并提供相应的设计文档内容；24%

许振东：任务二设计文档的的整理和书写；学习课本第七章listener和visitor的知识；完成第八章Building Some Real Language Applications 中**Loading CSV Data**的任务；19%

沈昊：学习课本第七章listener和visitor的知识；提供任务二会议纪要部分内容；完成第八章Building Some Real Language Applications 中**Translating JSON to XML**的任务；并提供相应的设计文档内容；19%

李研：任务二会议纪要第七章基础知识的整理与讲解；提供任务二会议纪要部分内容；19%

胡书豪：提供任务二会议纪要部分内容；完成第八章Building Some Real Language Applications 中**Generating a Call Graph**的任务；并提供相应的设计文档内容；19%

基础知识学习

Chapter 7. parse-tree listeners and visitors

任务二的第1次会议：2022.3.19 19:00-22:30 项目制3的第4次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），开启任务二，精读参考资料第七章。细致学习了 listener和visitor的原理，并做了详细的记录，为进入第八章正式开始任务二打基础。

第七章主要介绍了ANTLR自动生成的语法分析树遍历机制的工作方式和原理。

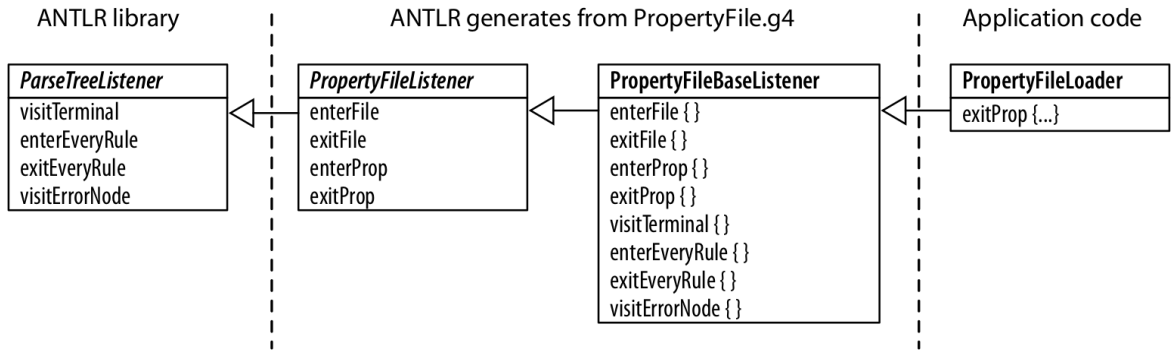
7.1 内嵌操作到listener的演进

7.1 中构建了两个不同的语言类应用程序，仅继承 ANTLR 自动生成的语法分析器类或仅实现新子类即可完成。但受限于内嵌动作，该语法仅能生成Java编写的语法分析器。为使语法可被重用并具有语言中立性，需要使用监听器和访问器来完全避免内嵌。

7.2 使用语法分析树监听器编写程序

7.2 介绍了使用语法分析树监听器编写程序的过程。构建应用逻辑和语法松耦合的语言类应用程序的关键在于，令语法分析器建立一棵语法分析树，然后在遍历该树的过程中触发应用逻辑代码。在得到语法分析树后，就可以使用ParserTreeWalker来访问它的全部节点并触发这些节点上的enter和exit方法。ANTLR自动生成了PropertyFileBaseListerner的默认实现，包含了所有方法的空实现，而这就让我们能够只覆盖那些我们所关心的方法。

示例的属性文件加载器中就在7.1的单个方法基础上引入了监听器机制（继承了监听器基类方法而非语法分析器，且监听器方法在语法分析器完成解析后被触发）。

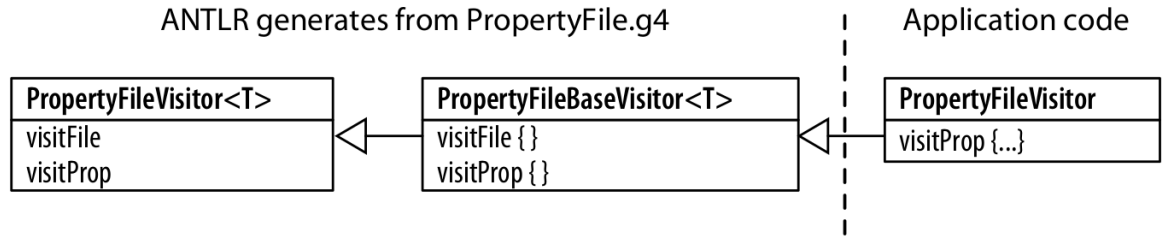


上图揭示了各接口与类之间的继承关系。

7.3 使用访问器编写应用程序

7.3介绍了使用访问器编写程序的过程。步骤如下：令ANTLR生成一个访问器接口，实现该接口，然后编写一个测试程序对语法分析树调用visit()方法，全过程中无需跟语法交互。

下图展示了与访问器相关的接口和类的继承关系。



进一步，访问器机制和监听器机制下的测试程序之间的最大区别在于：访问器机制里的测试程序不需要ParseTreeWalker，而是通过访问器来访问语法分析器生成的树。

7.4 有效的“注释”

7.4我们取了一个**新的subtitle**，虽然不如原书籍专业化，但是似乎是更好理解的。有时候在语法文件里，我们不知道选取哪个规则进行运用，就需要使用#运算符，为任意规则的最外层备选分支提供标签，像极了python代码加入注释增加可读性。之后ANTLR会为每个备选分支生成一个单独的监听器方法，这样就不需要再使用词法符号标签。

7.5 Share Info

7.5介绍了如何在事件中共享信息（传参、返回值），提供了三种方法。

第一种方法为使用访问器遍历语法分析树并返回值；第二种方法为使用栈来模拟返回值，将监听器的局部结果保存在一个成员变量中；第三种方法是将数据直接存储在语法分析树里，为每个节点添加标注（使用Map将任意值与节点一一对应）。

以方法三为例，下图为LExpr语法分析树：

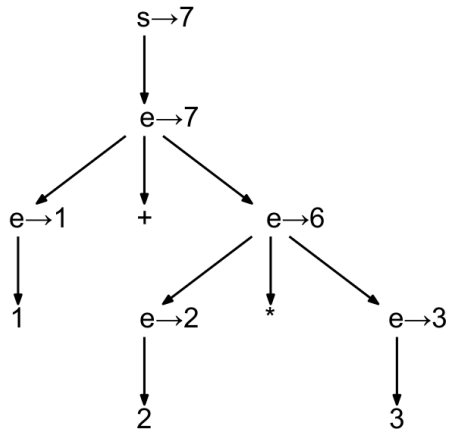


Figure 8—The LExpr grammar parse tree for 1+2*3

不同的方案具有不同的优缺点：使用访问器的代码具有良好的可读性，空间效率高，但所有值都必须具有相同的类型。基于栈的解决方案中可以传递多个参数值和返回值，同样具有较高空间效率，但在手工操作栈的过程中存在失误的可能性。树标注允许向事件方法提供任意信息来操纵语法分析树的各个节点，传递数据不容易失误，但因为局部结果会被保留，带来了更大的内存消耗。

Building Some Real Language Applications

任务二第2次会议：2022.3.20 19:00-22:30 项目制3的第5次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），小组继续任务二，实现8.1、8.2、8.3的任务，并跟进会议记录和设计文档。

1. Loading CSV Data

为获取精确的监听器方法，需要对任务一中完成的CSV语法的备选分支进行标记

修改后的CSV.g4文件如下：

```
grammar CSV;
file : hdr row+ ;
hdr : row ;
row : field (',' field)* '\r'? '\n' ;
field
  : TEXT    # text
  | STRING   # string
  |       # empty
  ;
TEXT : ~[,\n\r]+;
STRING : '"'('""'|~'"')* '"' ;  // quote-quote is an escaped quote
```

CSV语法监听器的实现呈现如下。代码中呈现了小组编写时对代码的理解。

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.FileInputStream;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.LinkedHashMap;
/**
```

```
 * Terence Parr, The Definitive ANTLR4 Reference, Chapter 8
 */
public class LoadCSV {
    public static class Loader extends CSVBaseListener {
        public static final String EMPTY = "";
        /**
         *这个列表中的每个元素是一个代表一行数据的MAP，该mao是从字段名到字段值的映射
         */
        List<Map<String, String>> rows = new ArrayList<Map<String, String>>();
        /**
         * 列名的列表
         */
        List<String> header;
        /**
         * 构造一个存放当前行中所有字段值的列表
         */
        List<String> currentRowFieldValues;
        /**
         * 提取合适的字符串，并将其加入currentRow-FieldValues
         */
        public void exitString(CSVParser.StringContext ctx) {
            currentRowFieldValues.add(ctx.STRING().getText());
        }
        public void exitText(CSVParser.TextContext ctx) {
            currentRowFieldValues.add(ctx.TEXT().getText());
        }
        public void exitEmpty(CSVParser.EmptyContext ctx) {
            currentRowFieldValues.add(EMPTY);
        }
        /**
         * 在第一行exitRow（）方法执行结束后，currentRowFieldValues就包含了全部的列名
         */
        public void exitHdr(CSVParser.HdrContext ctx) {
            header = new ArrayList<String>();
            header.addAll(currentRowFieldValues);
        }
        /**
         * 处理行数据
         * 这个过程需要两个操作：开始和结束对一行的操作
         * 当开始对一行的处理时，我们需要创建（清楚）currentRowFieldValues，以备接受后续数
据
         */
        public void enterRow(CSVParser.RowContext ctx) {
            currentRowFieldValues = new ArrayList<String>();
        }
        /**
         * 在完成对一行的处理时我们需要考虑上下文
         * 如果当前是一个数据行就创建map同步遍历header和currentRow-FieldValues，将映射关
系放入该map中
         */
        public void exitRow(CSVParser.RowContext ctx) {
            // 标题行什么都不做
            //if (ctx.parent instanceof CSVParser.HdrContext) return; OR:
            if (ctx.getParent().getRuleIndex() == CSVParser.RULE_hdr) return;
            // 以下为数据行
            Map<String, String> m = new LinkedHashMap<String, String>();
            int i = 0;
            for (String v : currentRowFieldValues) {
```

```
                m.put(header.get(i), v);
                i++;
            }
            rows.add(m);
        }
    }
    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if (args.length > 0) inputFile = args[0];
        InputStream is = System.in;
        if (inputFile != null) is = new FileInputStream(inputFile);
        CharStream input = CharStreams.fromStream(is, StandardCharsets.UTF_8);
        CSVLexer lexer = new CSVLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        CSVParser parser = new CSVParser(tokens);
        ParseTree tree = parser.file();
        //System.out.println(tree.toStringTree(parser));
        //进行打印数据行
        ParseTreeWalker walker = new ParseTreeWalker();
        Loader loader = new Loader();
        walker.walk(loader, tree);
        System.out.println(loader.rows);
    }
}
```

对于已给出的t.csv进行测试，t.csv如下

```
Details,Month,Amount
Mid Bonus,June,"$2,000"
,January,"""zippo"""
Total Bonuses,"","$5,000"
```

在终端输入如下命令:

```
antlr4 CSV.g4
javac CSV*.java LoadCSV.java
java LoadCSV t.csv
```

结果如下:

```
[{stuid=1, stuname=hyr, score=100}, {stuid=2, stuname=xzd, score=99}, {stuid=3,
stuname=ly, score=99}, {stuid=4, stuname=sh, score=99}, {stuid=}, {stuid=}]
```

进一步对任务一中自主设计的两个测试用例进行测试。

test1.csv:

```
stuid,stuname,score
1,hyr,100
2,xzd,99
3,ly,99
4,sh,99
```

操作步骤:

```
antlr4 CSV.g4
javac CSV*.java LoadCSV.java
java LoadCSV test1.csv
```

结果如下：

```
[{stuid=1, stuname=hyr, score=100}, {stuid=2, stuname=xzd, score=99}, {stuid=3,
stuname=ly, score=99}, {stuid=4, stuname=sh, score=99}, {stuid=}, {stuid=}]
```

test2.csv:

```
name,address,goods
hyr,"tju",soap
xzd,"tianjin""wuhu""",chiken
liyan,"daqing",coke
```

终端操作步骤：

```
antlr4 CSV.g4
javac CSV*.java LoadCSV.java
java LoadCSV test2.csv
```

结果如下：

```
[{name=hyr, address="tju", goods=soap}, {name=xzd, address="tianjin""wuhu""", go
ods=chiken}, {name=liyan, address="daqing", goods=coke}, {name=}]
```

2. Translating JSON to XML

按照要求，将语法文件修改如下：

```
#JSON语法的分支标记版
grammar JSON;
json
    : object
    | array
    ;
object
    : '{' pair (',' pair)* '}'    # AnObject
    | '{' '}'                     # EmptyObject
    ;
pair
    : STRING ':' value
    ;
array
    : '[' value (',' value)* ']'  # ArrayOfValues
    | '[' ']'                     # EmptyArray
    ;
value
    : STRING                      # String
    | NUMBER                      # Atom
    | object                      # ObjectValue
    | array                       # ArrayValue
    | 'true'                      # Atom
    | 'false'                     # Atom
    | 'null'                      # Atom
```

```
    ;
LCURLY : '{' ;
LBRACK : '[' ;
STRING
    : '"' (ESC | ~["\\])* '"' ;
fragment ESC : '\\' (["\\/bfnrt] | UNICODE) ;
fragment UNICODE : 'u' HEX HEX HEX HEX;
fragment HEX : [0-9a-fA-F] ;
NUMBER
    : '-'? INT '.' [0-9]+ EXP?    // 1.35, 1.35E-9, 0.3, -4.5
    | '-'? INT EXP                // 1e10 -2e4
    | '-'? INT                    // -3, 45
    ;
fragment INT : '0' | [1-9] [0-9]* ;   // no leading zeros
fragment EXP :  [Ee] [+\-]? INT ;
WS : [ \t\n\r]+ -> skip;
```

JSON2XML.java文件的实现如下。代码中呈现了小组编写时对代码的理解。

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import java.io.FileInputStream;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
public class JSON2XML {
    //标注语法分析树。将每棵树翻译完的字符串存储在子树的根节点中。
    public static class XMLEmitter extends JSONBaseListener {
        ParseTreeProperty<String> xml = new ParseTreeProperty<String>();
        String getXML(ParseTree ctx) {
            return xml.get(ctx);
        }
        void setXML(ParseTree ctx, String s) {
            xml.put(ctx, s);
        }
        //使用set将根元素object或array生成的结果标注语法分析树的根节点。
        public void exitJson(JSONParser.JsonContext ctx) {
            setXML(ctx, getXML(ctx.getChild(0)));
        }
    /*
    json的对象由键值对组成。因此对于每个object规则在anobject备选分支中的键值对，
    我们要将对应的XML追加到语法分析树中存储的结果之后。
    */
        public void exitAnObject(JSONParser.AnObjectContext ctx) {
            StringBuilder buf = new StringBuilder();
            buf.append("\n");
            for (JSONParser.PairContext pctx : ctx.pair()) {
                buf.append(getXML(pctx));
            }
            setXML(ctx, buf.toString());
        }
        public void exitEmptyObject(JSONParser.EmptyObjectContext ctx) {
            setXML(ctx, "");
        }
    //处理数组的方式与上相似，从各子节点中获取XML结果之后，放入<element>标签进行连接。
        public void exitArrayOfValues(JSONParser.ArrayOfValuesContext ctx) {
            StringBuilder buf = new StringBuilder();
            buf.append("\n");
```

```
            for (JSONParser.ValueContext vctx : ctx.value()) {
                buf.append("<element>");
                buf.append(getXML(vctx));
                buf.append("</element>");
                buf.append("\n");
            }
            setXML(ctx, buf.toString());
        }
        public void exitEmptyArray(JSONParser.EmptyArrayContext ctx) {
            setXML(ctx, "");
        }
    /*
    翻译完成value规则对应的所有元素后，需要处理键值对转换为标签和文本。
    开始和结束标签之间的文本来源于value子节点。
    */
        public void exitPair(JSONParser.PairContext ctx) {
            String tag = stripQuotes(ctx.STRING().getText());
            JSONParser.ValueContext vctx = ctx.value();
            String x = String.format("<%s>%s</%s>\n", tag, getXML(vctx), tag);
            setXML(ctx, x);
        }
    /*除了rule()方法匹配到一个对象或者数组，其可以将这些复合元素的翻译结果拷贝到
    自身的语法分析树的的节点中*/
        public void exitObjectValue(JSONParser.ObjectValueContext ctx) {
            // analogous to String value() { return object(); }
            setXML(ctx,getXML(ctx.object()));
        }
        public void exitArrayValue(JSONParser.ArrayValueContext ctx) {
            setXML(ctx, getXML(ctx.array()));  // String value() { return
array(); }
        }
    //Atom节点对应的标注值要与词法符号的文本内容相匹配。
        public void exitAtom(JSONParser.AtomContext ctx) {
            setXML(ctx, ctx.getText());
        }
    //字符串的处理基本和上述相同，除了需要额外剥离双引号
        public void exitString(JSONParser.StringContext ctx) {
            setXML(ctx, stripQuotes(ctx.getText()));
        }
        public static String stripQuotes(String s) {
            if ( s == null ||s.charAt(0) != '"') return s;
            return s.substring(1, s.length() - 1);
        }
    }
    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if (args.length > 0)
            inputFile = args[0];
        InputStream is = System.in;
        if (inputFile != null) {
            is = new FileInputStream(inputFile);
        }
        CharStream input = CharStreams.fromStream(is, StandardCharsets.UTF_8);
        JSONLexer lexer = new JSONLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        JSONParser parser = new JSONParser(tokens);
        parser.setBuildParseTree(true);
        ParseTree tree = parser.json();
```

```
        // show tree in text form
        System.out.println(tree.toStringTree(parser));
        ParseTreeWalker walker = new ParseTreeWalker();
        XMLEmitter xmlEmitter = new XMLEmitter();
        walker.walk(xmlEmitter, tree);
        System.out.println(xmlEmitter.getXML(tree));
    }
}
```

进一步，对于任务一中设计的两个json用力测试如下：

test1.json:

```
[1,2,3]
```

test2.json:

```
{
      "x1":  {
          "x2" : {
             "x3" : ["\u0001", "\u0002", "\u0003"],
             "x4" : 1.4e8,
             "x5" : -10
            },
          "x5" : false
      }
}
```

终端操作步骤：

```
antlr4 JSON.g4
javac JSON*.java
java JSON2XML test1.json
java JSON2XML test2.json
```

结果如下：

```
(py37) zima-blue@1:~/learning/Project3/tmp/json$ antlr4 JSON.g4
(py37) zima-blue@1:~/learning/Project3/tmp/json$ javac JSON*.java
(py37) zima-blue@1:~/learning/Project3/tmp/json$ java JSON2XML
JSON2XML$XMLEmitter.class              JSONParser$ArrayValueContext.class
JSON2XML.class                         JSONParser$AtomContext.class
JSON2XML.java                          JSONParser$EmptyArrayContext.class
JSONBaseListener.class                 JSONParser$EmptyObjectContext.class
JSONBaseListener.java                  JSONParser$JsonContext.class
JSON.g4                                JSONParser$ObjectContext.class
JSON.interp                            JSONParser$ObjectValueContext.class
JSONLexer.class                        JSONParser$PairContext.class
JSONLexer.interp                       JSONParser$StringContext.class
JSONLexer.java                         JSONParser$ValueContext.class
JSONLexer.tokens                       JSONParser.class
JSONListener.class                     JSONParser.java
JSONListener.java                      JSON.tokens
JSONParser$AnObjectContext.class       test1.json
JSONParser$ArrayContext.class          test2.json
JSONParser$ArrayOfValuesContext.class
(py37) zima-blue@1:~/learning/Project3/tmp/json$ java JSON2XML test1.json
(json (array [ (value 1) , (value 2) , (value 3) ]))

<element>1</element>
<element>2</element>
<element>3</element>

(py37) zima-blue@1:~/learning/Project3/tmp/json$ java JSON2XML test2.json
(json (object { (pair "x1" : (value (object { (pair "x2" : (value (object { (pair "x3" : (value (arra
y [ (value "\u0001") , (value "\u0002") , (value "\u0003") ])) , (pair "x4" : (value 1.4e8)) , (pair
 "x5" : (value -10)) })) , (pair "x5" : (value false)) })) })))

<x1>
<x2>
<x3>
<element>\u0001</element>
<element>\u0002</element>
<element>\u0003</element>
</x3>
<x4>1.4e8</x4>
<x5>-10</x5>
</x2>
<x5>false</x5>
</x1>
```

3. Generating a Call Graph

任务二第3次会议：2022.3.23 14:00-17:00 项目制3的第六次会议（参会人：胡轶然、沈昊、李研、胡书豪），小组继续任务二，实现任务8.4，完善检查会议纪要和设计文档，并开启任务三的摸索与第九章的学习。至此，任务二全部完成。

对任务一中Cymbol.g4中的一些备选分支进行标记，修改后如下：

```
grammar Cymbol ;
file   : (functionDecl | varDecl)+ ;
varDecl
  : type ID ('=' expr)? ';'
  ;
type   : 'float' | 'int' | 'void' ; // user-defined type
functionDecl
  : type ID '(' formalParameters? ')' block   // "void f(int x) {...}"
  ;
formalParameters
  : formalParameter (',' formalParameter)*
  ;
formalParameter
  : type ID
  ;
block : '{' stat* '}' ;   // possibly empty statement block
stat
  : block
  | varDecl
  | 'if' expr 'then' stat ('else' stat)?
```

```
  | 'return' expr? ';'
  | expr '=' expr ';'   // assignment
  | expr ';'        // func call
  ;
expr
  : ID '(' exprList? ')' # Call   // func call like f(), f(x), f(1,2)
  | ID '[' expr ']'    # Index   // array index like a[i], a[i][j]
  | '-' expr        # Negate  // unary minus
  | '!' expr        # Not    // boolean not
  | expr '*' expr     # Mult   // multiply
  | expr ('+'|'-') expr  # AddSub  //
  | expr '==' expr     # Equal   // equality comparison
  | ID           # Var
  | INT          # Int
  | '(' expr ')'      # Parens
  ;
exprList
  : expr (',' expr)* ;   // arg list
K_FLOAT : 'float';
K_INT  : 'int';
K_VOID : 'void';
ID        : LETTER (LETTER|DIGIT)* ;
fragment LETTER : [a-zA-Z\u0080-\u00FF] ;
INT       : '-'? DIGIT+ ;
fragment DIGIT : [0-9] ;
WS        : [ \t\n\r]+ -> skip;
LINE_COMMENT   : '//' .*? '\r'? '\n' -> skip; // Match "//" stuff '\n'
COMMENT     : '/*' .*? '*/'    -> skip; // Match "/*" stuff "*/"
```

调用图生成器程序的实现如下。代码中的注释呈现了小组编写时对代码的理解。

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.misc.*;
import org.antlr.v4.runtime.tree.*;

import java.io.FileInputStream;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;
import java.util.Set;

public class CallGraph {

    public static class Graph {
        //这里使用的是org.antlr.v4.runtime.misc: OrderedHashSet, MultiMap
        Set<String>nodes = new OrderedHashSet<String>();    //函数的列表
        MultiMap<String, String> edges =                    //调用者 -> 被调用者
            new MultiMap<String, String>();

        public void edge(String source, String target) {
            edges.map(source, target);
        } //边

        //toDOT() 完整地获取对应的DOT代码
        public String toDOT() {
            StringBuffer buf = new StringBuffer();
            buf.append("digraph G {\n");
            buf.append("    ranksep=.25\n");
```

```
            buf.append("    edge [arrowsize=.5]\n");
            buf.append("    node [shape=circle, fontname=\"ArialNarrow\",\n");
            buf.append("          fontsize=12, fixedsize=true, height=.45];\n");
            buf.append("    ");
            for (String node : nodes) { // 首先打印所有节点
                buf.append(node);
                buf.append("; ");
            }
            buf.append("\n");
            for (String src : edges.keySet()) {
                for (String trg : edges.get(src)) {
                    buf.append("    ");
                    buf.append(src);
                    buf.append(" -> ");
                    buf.append(trg);
                    buf.append(";\n");
                }
            }
            buf.append("}\n");
            return buf.toString();
        }
    }

    //使用监听器填充这些数据结构
    static class FunctionListener extends CymbolBaseListener {
        Graph graph = new Graph();
        String currentFunctionName = null;
        //监听器需要两个用于记录的字段

        public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
            currentFunctionName = ctx.ID().getText();
            graph.nodes.add(currentFunctionName);
        }//当语法分析器遇到函数定义时的方法，令其记录当前函数名

        public void exitCall(CymbolParser.CallContext ctx) {
            String funcName = ctx.ID().getText();
            // 将当前函数映射到被调用函数上
            graph.edge(currentFunctionName, funcName);
        }//当语法分析器发现函数调用时，程序就会记录一条从当前函数到被调用的函数的边
    }

    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if (args.length > 0) inputFile = args[0];
        InputStream is = System.in;
        if (inputFile != null) is = new FileInputStream(inputFile);
        CharStream input = CharStreams.fromStream(is, StandardCharsets.UTF_8);
        CymbolLexer lexer = new CymbolLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        CymbolParser parser = new CymbolParser(tokens);
        ParseTree tree = parser.file();
        //System.out.println(tree.toStringTree(parser));

        //在遍历中使用自定义的监听器，并产生期望的输出
        ParseTreeWalker walker = new ParseTreeWalker();
        FunctionListener collector = new FunctionListener();
        walker.walk(collector, tree);
        System.out.println(collector.graph.toString());
```

```
        System.out.println(collector.graph.toDOT());
    }
}
```

接下来对任务一自主设计用例进行测试。

test1.cymbol:

```
// Cymbol test1 by hyr
int gv = 6; // 全局变量声明
int a;
int plus(int a){
    return x+1; //加法
}
int fact(int x, int y) { // factorial function
if x==1 then return -x; //一元否定
if x==2 then return !x; //布尔非
if x==3 then return x*y; //乘法
if x==y then return (x+y)*2; //括号表达式
return x * plus(y);
}
```

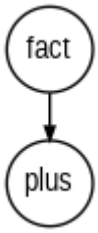
终端操作步骤:

```
antlr4 Cymbol.g4
javac Cymbol*.java CallGraph.java
java CallGraph test1.cymbol
```

结果如下:

```
CallGraph$Graph@497470ed
digraph G {
    ranksep=.25
    edge [arrowsize=.5]
    node [shape=circle, fontname="ArialNarrow",
          fontsize=12, fixedsize=true, height=.45];
    plus; fact;
    fact -> plus;
}
```

在graphviz中预览函数调用图:



4. Validating Program Symbol Usage

为Cymbol构造符号表，用于检测未定义的变量和函数，确保变量和函数被正确运用。编写的Cymbol验证器需要作出以下校验:

1.引用的变量必须有可以见的（在定义域中）定义;

2.引用的函数必须有定义（函数必须以任何顺序出现）;

3.变量不可用作函数;

4.函数不可用作变量;

在书本的实例中，vars.cymbol与vars2.cymbol中部分标识符无效或存在冲突:

```
vars.cymobl
int f(int x, float y) {
  g();           // forward reference is ok
  i = 3;         // no declaration for i (error)
  g = 4;         // g is not variable (error)
  return x + y;  // x, y are defined, so no problem
}
void g() {
  int x = 0;
  float y;
  y = 9;         // y is defined
  f();           // backward reference is ok
  z();           // no such definition (error)
  y();           // y is not function (error)
  x = f;         // f is not a variable (error)
}

vars2.cymbol
int x;
int y;
void a()
{
  int x;
  x = 1;    // x resolves to current scope, not x in globalscope
  y = 2;    // y is not found in current scope, but resolves in global
  { int y = x; }
}
void b(int z)
{}
```

首先定义验证器的基本结构DefPhase.java。文件的实现如下。代码中呈现了小组编写时对代码的理解。

```
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.tree.ParseTreeProperty;

public class DefPhase extends CymbolBaseListener {

    ParseTreeProperty<Scope> scopes = new ParseTreeProperty<Scope>();
    GlobalScope globals;
    Scope currentScope;    // 当前符号的作用域

    public void enterFile(CymbolParser.FileContext ctx) {
        globals = new GlobalScope(null);
        currentScope = globals;
    }

    public void exitFile(CymbolParser.FileContext ctx) {
        System.out.println(globals);
    }
```

```
    public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
        String name = ctx.ID().getText();
        int typeTokenType = ctx.type().start.getType();
        Symbol.Type type = CheckSymbols.getType(typeTokenType);
        // 新建一个指向外围作用域的作用域，这样就完成了入栈操作
        FunctionSymbol function = new FunctionSymbol(name, type, currentScope);
        currentScope.define(function);  // 在当前作用域中定义函数
        saveScope(ctx, function);       // 入栈：设置函数作用域的父作用域为当前作用域
        currentScope = function;        // 现在当前作用域就是函数作用域了
    }

    void saveScope(ParserRuleContext ctx, Scope s) {
        scopes.put(ctx, s);
    }

    public void exitFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
        System.out.println(currentScope);
        currentScope = currentScope.getEnclosingScope();  // 作用域出栈
    }

    public void enterBlock(CymbolParser.BlockContext ctx) {
        // push new local scope
        currentScope = new LocalScope(currentScope);
        saveScope(ctx, currentScope);
    }

    public void exitBlock(CymbolParser.BlockContext ctx) {
        System.out.println(currentScope);
        currentScope = currentScope.getEnclosingScope(); // pop scope
    }

    public void exitFormalParameter(CymbolParser.FormalParameterContext ctx) {
        defineVar(ctx.type(), ctx.ID().getSymbol());
    }

    public void exitVarDecl(CymbolParser.VarDeclContext ctx) {
        defineVar(ctx.type(), ctx.ID().getSymbol());
    }

    void defineVar(CymbolParser.TypeContext typeCtx, Token nameToken) {
        int typeTokenType = typeCtx.start.getType();
        Symbol.Type type = CheckSymbols.getType(typeTokenType);
        VariableSymbol var = new VariableSymbol(nameToken.getText(), type);
        currentScope.define(var);   // 在当前作用域中定义符号
    }
}
```

总验证器CheckSymbol.java文件的实现如下。代码中呈现了小组编写时对代码的理解。

```
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.tree.*;

import java.io.FileInputStream;
```

```
import java.io.InputStream;
import java.nio.charset.StandardCharsets;

public class CheckSymbols {

    public static Symbol.Type getType(int tokenType) {
        switch (tokenType) {
        case CymbolParser.K_VOID :  return Symbol.Type.tVOID;
        case CymbolParser.K_INT :   return Symbol.Type.tINT;
        case CymbolParser.K_FLOAT : return Symbol.Type.tFLOAT;
        }//数据类型

        return Symbol.Type.tINVALID;
    }

    public static void error(Token t, String msg) {
        System.err.printf("line %d:%d %s\n", t.getLine(),
t.getCharPositionInLine(), msg);
    }

    public static void main(String[] args) throws Exception {
        String inputFile = null;
        if (args.length > 0)
            inputFile = args[0];
        InputStream is = System.in;
        if (inputFile != null) {
            is = new FileInputStream(inputFile);
        }
        CharStream input = CharStreams.fromStream(is, StandardCharsets.UTF_8);
        CymbolLexer lexer = new CymbolLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        CymbolParser parser = new CymbolParser(tokens);
        parser.setBuildParseTree(true);
        ParseTree tree = parser.file();
        // 以文本形式显示树
        System.out.println(tree.toStringTree(parser));

        ParseTreeWalker walker = new ParseTreeWalker();
        DefPhase def = new DefPhase();
        walker.walk(def, tree);
        // 新建一个阶段，将def中的符号表信息传递给该阶段
        RefPhase ref = new RefPhase(def.globals, def.scopes);
        walker.walk(ref, tree);
    }
}
```

小组编写和设计如下测试用例vars_test_error.cymbol:

```
float A (int x, int y) {
        B(x);     // B中未定义引用内容
        int z = 1;     // 定义无误
        B = x - z;    // B不是变量
        return B;    // 返回类型错误
}
void B () {
        C();       // 未定义函数C
        c = C();        // 未定义变量c，且赋值错误
```

```
        int m = 1;          // 定义无误
        int n = 2;          // 定义无误
        float l;            // 定义无误
        l = m + n;         // 运算无误
        l = A ();         // A不是变量，赋值错误
}
```

终端操作步骤：

```
antlr4 Cymbol.g4
javac Cymbol*.java CheckSymbols.java *Phase.java *Scope.java *Symbol.java
java CheckSymbols vars_test_error.cymbol
```

结果如下：

```
listener$ java CheckSymbols vars_test_error.cymbol
(file (functionDecl (type float) A ( (formalParameters (formalParameter (type in
t) x) , (formalParameter (type int) y)) ) (block { (stat (expr B ( (exprList (ex
pr x)) )) ;) (stat (varDecl (type int) z = (expr 1) ;)) (stat (expr B) = (expr (
expr x) - (expr z)) ;) (stat return (expr B) ;) })) (functionDecl (type void) B
( ) (block { (stat (expr C ( )) ;) (stat (expr c) = (expr C ( )) ;) (stat (varDe
cl (type int) m = (expr 1) ;)) (stat (varDecl (type int) n = (expr 2) ;)) (stat
(varDecl (type float) l ;)) (stat (expr l) = (expr (expr m) + (expr n)) ;) (stat
 (expr l) = (expr A ( )) ;) }))
locals:[z]
function<A:tFLOAT>:[<x:tINT>, <y:tINT>]
locals:[m, n, l]
function<B:tVOID>:[]
globals:[A, B]
line 4:8 B is not a variable
line 5:15 B is not a variable
line 9:8 no such function: C
line 10:8 no such variable: c
line 10:12 no such function: C
```

至此任务二全部完成。