

# 天津大学

## 编译原理实践



组 长： 朱相举 (3019244319)

组 员： 李润泽 (3019244266)

遆铮 (3019244106)

吴俊杰 (3019207400)

高建鸿 (3019244316)

任课教师： 胡静老师

2022 年 9 月 17 日

## 8.1 loading CSV

### 1) 转化为 MAP 的 LIST

我们的目标是写一个自定义的监视器——能够将 CSV 文件中数据加载到一种“由 MAP 组成的 LIST”中。所以将文件，转化为 一行行的 MAP，如

```
listeners/t.csv
Details,Month,Amount
Mid Bonus,June,"$2,000"
,January,"""zippo""""
Total Bonuses,"","$5,000"
```

转为如下三个 MAP:

```
[{Details=Mid Bonus, Month=June, Amount="$2,000"},
 {Details=, Month=January, Amount="""zippo"""},
 {Details=Total Bonuses, Month="", Amount="$5,000"}]
```

### 2) 构建数据结构关系

首先定义一个核心数据结构 rows;

其次需要一组列名，我们需要对 headrow 进行处理，并且放入一个临时列表中;

临时列表不妨命名为——currentRowFieldValues，其中与列名字段值相关。

### 3) 监视器部分

```
18 public class LoadCSV {
19     public static class Loader extends CSVBaseListener {
20         public static final String EMPTY = "";
21         /** Load a list of row maps that map field name to value */
22         List<Map<String,String>> rows = new ArrayList<Map<String, String>>();
23         /** List of column names */
24         List<String> header;
25         /** Build up a list of fields in current row */
26         List<String> currentRowFieldValues;
```

使用三种规则方法处理字段值，exitString, exitText, exitEmpty: 提取合适的

字符串,并将其加入 `currentRowFieldValues`(我们不妨暂时假设在首行的 `exitRow` 方法结束后, `currentRowFieldValues` 包含了所有列名)。

标题行要区分对待于普通行,所以要检查上下文,如果我们刚刚出力是标题行,那就什么也不做,因为列名不是数据。

在 `exitRow` 方法中,我们可以通过查看父节点的 `getRuleIndex()`的返回值得知当前的上下文,或者查看父节点的类型是否是 `HdrContext`。

如果当前行是数据行的话就创建 `MAP`, 同步遍历 `header` 中的列名和 `currentRowFieldValues` 中的字段值, 将映射关系放入 `MAP`。

```
37     public void exitRow(CSVParser.RowContext ctx) {
38         // If this is the header row, do nothing
39         // if ( ctx.parent instanceof CSVParser.HdrContext ) return; OR:
40         if ( ctx.getParent().getRuleIndex() == CSVParser.RULE_hdr ) return;
41         // It's a data row
42         Map<String, String> m = new LinkedHashMap<String, String>();
43         int i = 0;
44         for (String v : currentRowFieldValues) {
45             m.put(header.get(i), v);
46             i++;
47         }
48         rows.add(m);
49     }
```

## 4) 读入数据后打印

在使用 `ParseTree-Walker` 完成对语法分析树的遍历后, 使用 `LoadCSV` 类中的 `main()`就能打印出数据。

```
64     public static void main(String[] args) throws Exception {
65         String inputFile = null;
66         if ( args.length>0 ) inputFile = args[0];
67         InputStream is = System.in;
68         if ( inputFile!=null ) is = new FileInputStream(inputFile);
69         CSVLexer lexer = new CSVLexer(new ANTLRInputStream(is));
70         CommonTokenStream tokens = new CommonTokenStream(lexer);
71         CSVParser parser = new CSVParser(tokens);
72         parser.setBuildParseTree(true); // tell ANTLR to build a parse tree
73         ParseTree tree = parser.file();
74
75         ParseTreeWalker walker = new ParseTreeWalker();
76         Loader loader = new Loader();
77         walker.walk(loader, tree);
78         System.out.println(loader.rows);
79     }
```

## 5) 测试

```
LoadCSV.java:9: 警告: [deprecation] org antlr.v4.runtime中的ANTLRInputStream已过时
import org antlr.v4.runtime.ANTLRInputStream;

LoadCSV.java:69: 警告: [deprecation] org antlr.v4.runtime中的ANTLRInputStream已过时
    CSVLexer lexer = new CSVLexer(new ANTLRInputStream(is));

2 个警告
[[{"Details=Mid Bonus, Month=June, Amount="$2,000"}, {"Details=, Month=January, Amount:
""zippo""}, {"Details=Total Bonuses, Month="", Amount="$5,000"}]
```

编译过程使用了 `-Xlint:deprecation` 命令来处理过时的 API

加载 CSV 成功

## 8.2 Translating JSON to XML

和 CSV 语法一样，让我们首先对 JSON 语法中的备选分支做一定的标记，以便 ANTLR 生成更精确的监听器方法。

```
object
    :   '{' pair (',' pair)* '}'      # AnObject
    |   '{' '}'                        #
EmptyObject
    ;

array
    :   '[' value (',' value)* ']'    #
```

我们会用同样的方法处理 `value` 规则，不过稍微做出了一些改变。除了其中三个备选分支之外，其他都必须返回它匹配到的文本值，因此，我们可以对它们进行同样的标记，使得语法分析树遍历器为这些备选分支触发相同的监听器事件。

```

value
    :   STRING      # String
    |   NUMBER      # Atom
    |   object      # ObjectValue
    |   array        # ArrayValue
    |   'true'       # Atom
    |   'false'      # Atom
    |   'null'       # Atom
    ;

```

我们的翻译器的实现需要令每条规则返回与它匹配到的输入文本等价的 XML。为跟踪这些局部结果，我们将会使用 `xml` 字段和两个辅助方法来对语法分析树进行标注。

```

public class JSON2XML {
    public static class XMLEmitter extends
JSONBaseListener {
        ParseTreeProperty<String> xml = new
ParseTreeProperty<String>();
        String getXML(ParseTree pt) { return

```

我们会将每棵子树翻译完的字符串存储在该子树的根节点中。这样,工作在语法分析树更高层节点上的方法就能够获得它们,从而构造出更大的字符串。语法分析树的根节点中存储的字符串就是最终的翻译结果。

让我们从最简单的翻译开始。`value` 规则中的 `Atom` 备选分支“返回”(也就是在 `Atom` 节点对应的标注值)它匹配的词法符号中的文本内容(`ctx.getText()`获得对应规则匹配到的文本)。

```

public void exitAtom(JSONParser.AtomContext ctx) {
    setXML(ctx, ctx.getText());
}

```

除了需要额外剥离双引号之外(`stripQuotes` 是该文件提供的辅助方法),对字符串的处理基本和上述过程基本相同。

```

public void exitString(JSONParser.StringContext ctx) {
    setXML(ctx, stripQuotes(ctx.getText()));
}

```

如果 `rule 0` 方法匹配到的是一个对象或者一个数组,它就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节点中,下列代码给出了实现细节:

```

public void exitObjectValue(JSONParser.ObjectValueContext
ctx) {
    // analogous to String value() {return object();}
    setXML(ctx, getXML(ctx.object()));
}

```

在翻译完成 `value` 规则对应的所有元素后,我们需要处理键值对,将它们转换为标签和文本。生成的 XML 的标签名来源于 `STRING: 'value'` 备选分支中的 `STRING`。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);
}

```

JSON 的对象由一系列键值对组成。因此，对于每个 `object` 规则在 `AnObject` 备选分支中发现的键值对，我们将其对应的 XML 追加到语法分析树中储存的结果之后。

```
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) {
```

处理数组的方式与之相似，从各子节点中获取 XML 结果，将其分别放入 `<element>` 标签之后连接起来即可。

```

        public                                     void
exitArrayOfValues(JSONParser.ArrayOfValuesContext ctx) {
    StringBuilder buf = new StringBuilder();
    buf.append("\n");
    for (JSONParser.ValueContext vctx : ctx.value())
    {
        buf.append("<element>");    //  conjure  up
        element for valid XML
        buf.append(getXML(vctx));
        buf.append("</element>");
        buf.append("\n");
    }
    setXML(ctx, buf.toString());
}

        public                                     void
exitEmptyArray(JSONParser.EmptyArrayContext ctx) {
    setXML(ctx, "");
}

```

最后,我们需要用最终的结果——由根元素 `object` 或者 `array` 生成的结果——标注语法分析树的根节点。

```

json:   object
      |   array
      ;

```

下面是构建和测试的步骤:



t.json - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
{
  "description": "An imaginary server config file",
  "logs": {"level": "verbose", "dir": "/var/log"},
  "host": "antlr.org",
  "admin": ["parrt", "tombu"],
  "aliases": []
}
```

```
<description>An imaginary server config file</description>
<logs>
<level>verbose</level>
<dir>/var/log</dir>
</logs>
<host>antlr.org</host>
<admin>
<element>parrt</element>
<element>tombu</element>
</admin>
<aliases></aliases>
```

## 8.2 解析 DOT 语言

### 1. 完成 DOT 语法

```
1 grammar DOT;
2
3 graph.....: STRICT? (GRAPH | DIGRAPH) id? '{' stmt_list '}' ;
4 stmt_list.....: ( stmt ';' ? ) * ;
5 stmt.....: node_stmt
6 |.....: edge_stmt
7 |.....: attr_stmt
8 |.....: id '=' id
9 |.....: subgraph
10 |.....: ;
11 attr_stmt.....: (GRAPH | NODE | EDGE) attr_list ;
12 attr_list.....: ( '[' a_list? ']' ) + ;
13 a_list.....: ( id ( '=' id ) ? ',' ? ) + ;
14 edge_stmt.....: ( node_id | subgraph ) edgeRHS attr_list ? ;
15 edgeRHS.....: ( edgeop ( node_id | subgraph ) ) + ;
16 edgeop.....: '->' | '--' ;
17 node_stmt.....: node_id attr_list ? ;
18 node_id.....: id port ? ;
19 port.....: ':' id ( ':' id ) ? ;
20 subgraph.....: ( SUBGRAPH id ? ) ? '{' stmt_list '}' ;
21 id.....: ID
22 |.....: STRING
23 |.....: HTML_STRING
24 |.....: NUMBER
25 |.....: ;
26
27 STRICT.....: [Ss][Tt][Rr][Ii][Cc][Tt] ;
28 GRAPH.....: [Gg][Rr][Aa][Pp][Hh] ;
29 DIGRAPH.....: [Dd][Ii][Gg][Rr][Aa][Pp][Hh] ;
30 NODE.....: [Nn][Oo][Dd][Ee] ;
31 EDGE.....: [Ee][Dd][Gg][Ee] ;
32 SUBGRAPH.....: [Ss][Uu][Bb][Gg][Rr][Aa][Pp][Hh] ;
33
34 NUMBER.....: '-' ? ( '.' DIGIT+ | DIGIT+ ( '.' DIGIT* ) ? ) ;
35 fragment
36 DIGIT.....: [0-9] ;
37
38 STRING.....: '"' ( '\\' | . ) * ? '"' ;
39
40 ID.....: LETTER ( LETTER | DIGIT ) * ;
41 fragment
42 LETTER.....: [a-zA-Z\u0080-\u00FF_] ;
43
44 HTML_STRING.....: '<' ( TAG | ~[<>] ) * '>' ;
45 fragment
46 TAG.....: '<' .*? '>' ;
47
48 COMMENT.....: '/' * .*? '/' -> skip ;
49 LINE_COMMENT.....: '/' / .*? '\r'? '\n' -> skip ;
50
51 PREPROC.....: '#' .*? '\n' -> skip ;
52
53 WS.....: [ \t\n\r]+ -> skip ;
54
55
```

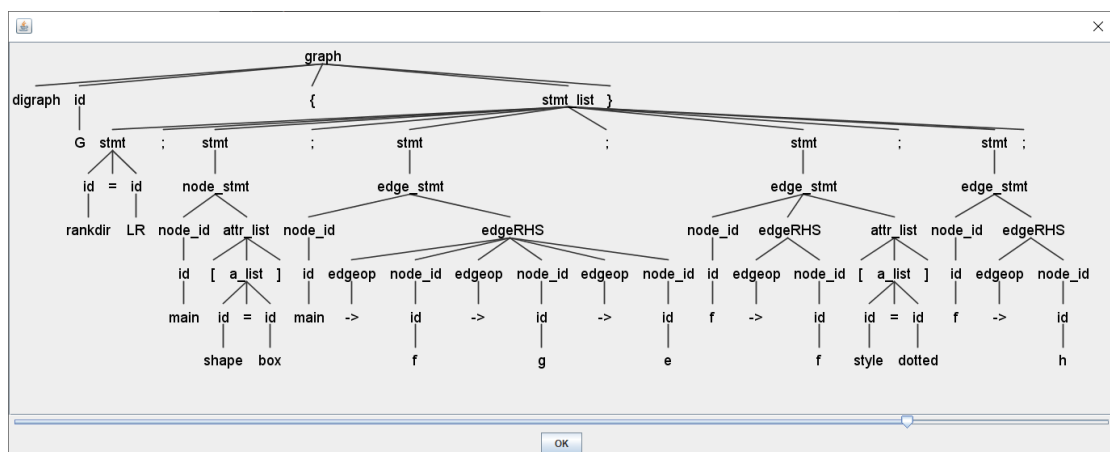


## 6. 其他两个测试用例

### 用例 1

```
DOT > ≡ 1.dot
1  digraph G {
2      ....rankdir=LR;
3      ....main [shape=box];
4      ....main -> f -> g -> e;
5      ....f -> f [style=dotted];
6      ....f -> h;
7  }
8
```

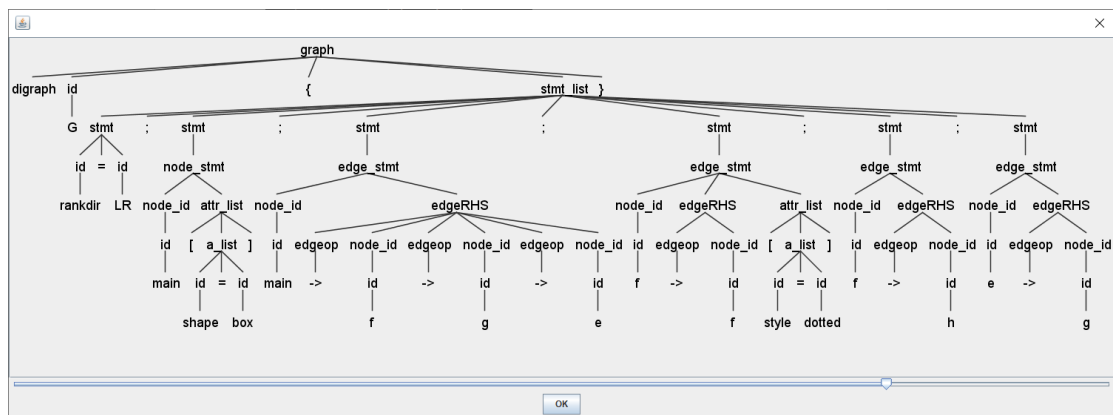
```
y:\antlr\antlr-4.0-complete.jar; org.antlr.v4.runtime.misc.TestRig DOT graph -tree 1.dot
(graph digraph (id G) { (stmt_list (stmt (id rankdir) = (id LR)) ; (stmt (node_stmt (node_id (id main)) (attr_list [ (a_list (id shape) = (id box)) ]))) ; (s
stmt (edge_stmt (node_id (id main)) (edgeRHS (edgeop ->) (node_id (id f)) (edgeop ->) (node_id (id g)) (edgeop ->) (node_id (id e))))) ; (stmt (edge_stmt (nod
e_id (id f)) (edgeRHS (edgeop ->) (node_id (id f)) (attr_list [ (a_list (id style) = (id dotted)) ]))) ; (stmt (edge_stmt (node_id (id f)) (edgeRHS (edgeop
->) (node_id (id h))))) ; ) })
```



## 用例 2

```
DOT > ≡ 2.dot
1  digraph G {
2      ... rankdir=LR;
3      ... main [shape=box];
4      ... main -> f -> g -> e;
5      ... f -> f [style=dotted];
6      ... f -> h;
7      ... e -> g
8  }
```

```
v\antlr\antlr-4.0-complete.jar; org.antlr.v4.runtime.misc.TestRig DOT graph -tree 2.dot
(graph digraph (id G) { (stmt_list (stmt (id rankdir) = (id LR)) ; (stmt (node_stmt (node_id (id main)) (attr_list [ (a_list (id shape) = (id box)) ]))) ; (
stmt (edge_stmt (node_id (id main)) (edgeRHS (edgeop ->) (node_id (id f)) (edgeop ->) (node_id (id g)) (edgeop ->) (node_id (id e))))) ; (stmt (edge_stmt (no
e_id (id f)) (edgeRHS (edgeop ->) (node_id (id f)) (attr_list [ (a_list (id style) = (id dotted)) ]))) ; (stmt (edge_stmt (node_id (id f)) (edgeRHS (edgeop
->) (node_id (id h))))) ; (stmt (edge_stmt (node_id (id e)) (edgeRHS (edgeop ->) (node_id (id g))))) } }
```



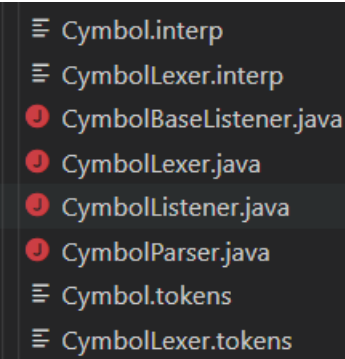
## 8.3 Generating a Call Graph

### 1. 完成 Cymbol 语法

```
1  grammar Cymbol;
2
3  file: ... (functionDecl | varDecl)+ ;
4
5  varDecl
6  | ... : ... type ID ('=' expr)? ';'
7  | ... ;
8  type: ... 'float' | 'int' | 'void' ;
9
10 functionDecl
11 | ... : ... type ID '(' formalParameters? ')' block
12 | ... ;
13
14 formalParameters
15 | ... : ... formalParameter (',' formalParameter)*
16 | ... ;
17 formalParameter
18 | ... : ... type ID
19 | ... ;
20
21 block: ... '{' stat* '}' ;
22
23 stat: ... block
24 | ... | ... varDecl
25 | ... | ... 'if' expr 'then' stat ('else' stat)?
26 | ... | ... 'return' expr? ';'
27 | ... | ... expr '=' expr ';'
28 | ... | ... expr ';'
29 | ... ;
30
31
32 expr: ... ID '(' exprList? ')' .....# Call
33 | ... | ... expr '[' expr ']' .....# Index
34 | ... | ... '-' expr .....# Negate
35 | ... | ... '!' expr .....# Not
36 | ... | ... expr '*' expr .....# Mult
37 | ... | ... expr ('+' | '-') expr .....# AddSub
38 | ... | ... expr '==' expr .....# Equal
39 | ... | ... ID .....# Var
40 | ... | ... INT .....# Int
41 | ... | ... '(' expr ')' .....# Parens
42 | ... ;
43
44 exprList : expr (',' expr)* ;
45
46 K_FLOAT : 'float';
47 K_INT : 'int';
48 K_VOID : 'void';
49 ID : ... LETTER (LETTER | [0-9])* ;
50 fragment
51 LETTER : [a-zA-Z] ;
52
53 INT : ... [0-9]+ ;
54
55 WS : ... [ \t\n\r]+ -> skip ;
56
57 SL_COMMENT
58 | ... : ... '/*' .*? '\n' -> skip
59 | ... ;
60
```

## 2. 生成 java 代码




























```
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>antlr4 Cymbol.g4  
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>java -cp C:\env\antlr\antlr-4.9.2-complete.jar;.;C:\env\jdk1.8.0_333\tools.jar;C:\env\jdk1.8.0_333\dt.jar;C:\env\antlr\antlr-4.9.2-complete.jar; org.antlr.v4.Tool Cymbol.g4  
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>
```



- ≡ Cymbol.interp
- ≡ CymbolLexer.interp
- 🔴 CymbolBaseListener.java
- 🔴 CymbolLexer.java
- 🔴 CymbolListener.java
- 🔴 CymbolParser.java
- ≡ Cymbol.tokens
- ≡ CymbolLexer.tokens

## 3. 编译 java 代码

```
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>javac Cymbol*.java CallGraph.java  
注: CallGraph.java使用或覆盖了已过时的 API。  
注: 有关详细信息, 请使用 -Xlint:deprecation 重新编译。  
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>
```

-  CallGraph\$FunctionListener.class
-  CallGraph\$Graph.class
-  CallGraph.class
-  CymbolBaseListener.class
-  CymbolLexer.class
-  CymbolListener.class
-  CymbolParser\$AddSubContext.class
-  CymbolParser\$BlockContext.class
-  CymbolParser\$CallContext.class
-  CymbolParser\$EqualContext.class
-  CymbolParser\$ExprContext.class
-  CymbolParser\$ExprListContext.class
-  CymbolParser\$FileContext.class
-  CymbolParser\$FormalParameterContext.class
-  CymbolParser\$FormalParametersContext.class
-  CymbolParser\$FunctionDeclContext.class
-  CymbolParser\$IndexContext.class
-  CymbolParser\$IntContext.class
-  CymbolParser\$MultContext.class
-  CymbolParser\$NegateContext.class
-  CymbolParser\$NotContext.class
-  CymbolParser\$ParensContext.class
-  CymbolParser\$StatContext.class
-  CymbolParser\$TypeContext.class
-  CymbolParser\$VarContext.class
-  CymbolParser\$VarDeclContext.class
-  CymbolParser.class

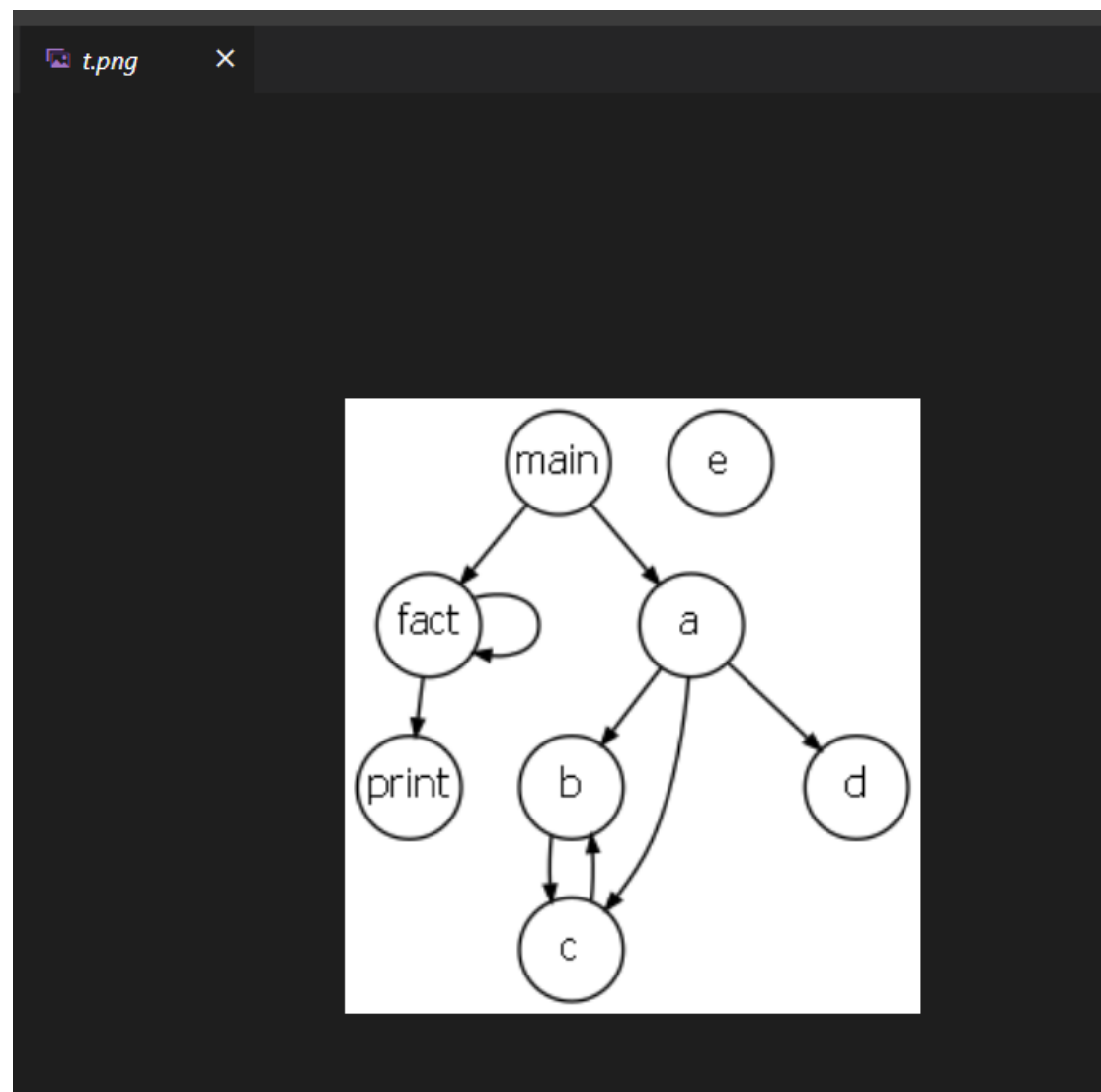
## 4. 运行

```
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>java CallGraph t.cymbol
edges: {main=[fact, a], fact=[print, fact], a=[b, c, d], b=[c], c=[b]}, functions: [main, fact, a, b, c, d, e]
digraph G {
    ranksep=.25;
    edge [arrowsize=.5]
    node [shape=circle, fontname="ArialNarrow",
        fontsize=12, fixedsize=true, height=.45];
    main; fact; a; b; c; d; e;
    main -> fact;
    main -> a;
    fact -> print;
    fact -> fact;
    a -> b;
    a -> c;
    a -> d;
    b -> c;
    c -> b;
}
```



## 5. 使用 Graphviz 生成调用图

```
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>dot -Tpng t.dot -o t.png  
(dot.exe:6432): Pango-WARNING **: couldn't load font "ArialNarrow Not-Rotated 12", falling back to "Sa  
ns Not-Rotated 12", expect ugly output.  
C:\Users\LuShengcan\Desktop\antlr-4.9.2\chapter8\Cymbol>
```



## 8.4 Validating Program Symbol Usage

### 1. 章节背景与目的

在为类似 Cymbol 的编程语言编写解释器、编译器或者翻译器之前，我们需要确保 Cymbol 程序中使用的符号（标识符）用法正确。在本节中，我们计划编写一个能做出以下校验的 Cymbol 验证器：

- 引用的变量必须有可见的（在作用域中）定义
- 引用的函数必须有定义（函数可以以任何顺序出现，即函数定义提升）
- 变量不可用作函数
- 函数不可用作变量

要满足以上全部条件，我们需要做一点工作，因此理解本例可能会花费比其他例子更多的时间。不过，我们的收获将为编写真实的语言处理工具奠定坚实基础。

让我们首先来看一些包含不同标识符引用的样例代码，其中一些标识符是无效的。

### 2. 符号表

我们通常称存储符号的数据结构为：“符号表”。实现这样的语言，要建议复杂的符号表结构。如果语言允许统一标识符在不同的上下文中是不同含义，那么就需要将符号按照作用域分组。

“作用域”：是一组符号的集合，例如，一组函数的参数列表，或者全局作用域中定义的变量和函数。

“符号表” 仅仅定义符号的仓库，不进行任何验证工作。

按照之前确定的规则，检查表达式中的变量和函数，以完成代码的验证，符号验证中需要两种基本操作——定义符合+解析符号。

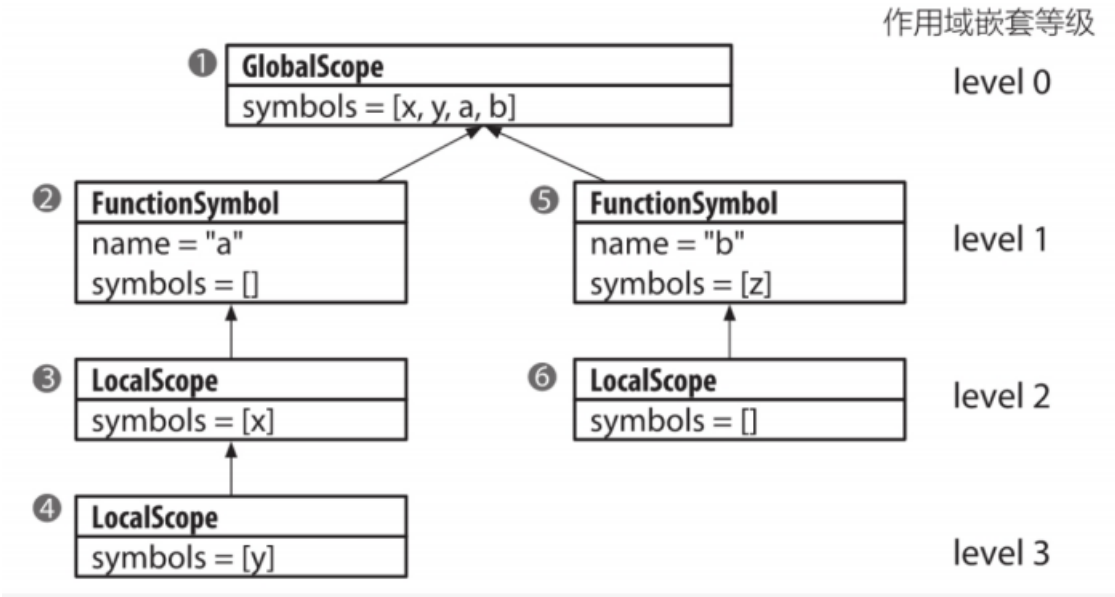
定义符号：将它添加到作用域中。

解析符号：确定该符号引用了哪种定义。

解析符号：意味着寻找最接近的符号定义，最接近的定义域就是最内层的代码块。

由于我们无法避免在同一集合中处理所有标识符时的冲突问题，这就是作用域的意义。

我们维护一组作用域，一个作用域中 一个标识符只允许被定义一次。我们还未每个作用域维护一个指向父作用域的指针。这样符号表就可以着手编写验证器了。



### 3.验证器的架构

验证器问题分解为“定义+解析”。

定义，需要“监听变量+函数定义”的事件，生成 Symbol 对象并加入作用域内。在函数定义开始时候，需要将一个新的作用域入栈，结束时，将该作用域出栈。

对于解析和验证符号引用，我们需要 监听表达式中的变量和符号引用 的事件。对于每个引用要验证是否存在匹配的符号定义，以及是否正确使用符号。

有一个难题，一个 Cymbol 程序在函数声明前就可以调用，称为“前向引用”可以通过两趟遍历支持：

- 第一遍历，对函数在内的符号定义
- 第二遍里，看到文件中全部的函数

定义阶段会创建很多作用域，需要保持对这些定义域的引用，否则垃圾回收器会将他们清除。

为了保证符号表从“定义” $\Rightarrow$ “解析”这个转换过程中始终存在，需要追踪这些作用域。

存储在语法分析树本身，或者使用一个 Map 将节点和值映射，以此方便分析一个引用对应的作用域。

## 4.定义和解析符号

### 4.1 定义阶段

确定全局的策略，开始编写验证器。从 DefPhase 开始，需要：

- 1.全局作用域的引用
- 2.一个追踪我们创建作用域的语法分析树标注器
- 3.一个指向当前作用域的指针

enterFile 启动整个验证过程，创建全局作用域；而 exitFile 负责打印结果。

```
13 public class DefPhase extends CymbolBaseListener {
14     ParseTreeProperty<Scope> scopes = new ParseTreeProperty<Scope>();
15     GlobalScope globals;
16     Scope currentScope; // define symbols in this scope
17     public void enterFile(CymbolParser.FileContext ctx) {
18         globals = new GlobalScope(null);
19         currentScope = globals;
20     }
21
22     public void exitFile(CymbolParser.FileContext ctx) {
23         System.out.println(globals);
24     }
25 }
```

当语法分析器发现一个函数定义，传概念一个 FunctionSymbol 对象，用来

- 1.作为一个符号；2.作为一个含参数的作用域。

为构建一个嵌套在全局作用域中的函数作用域，我们将函数作用域“入栈”！

“入栈”是通过把当前作用域设置为该函数作用域的父作用域，并将他本身设置为当前作用域完成的。

```

26     public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
27         String name = ctx.ID().getText();
28         int typeTokenType = ctx.type().start.getType();
29         Symbol.Type type = CheckSymbols.getType(typeTokenType);
30
31         // 新建一个作用域，定义父子关系，“入栈”
32         FunctionSymbol function = new FunctionSymbol(name, type, currentScope);
33         currentScope.define(function);
34         saveScope(ctx, function);
35         currentScope = function;
36     }

```

方法 SaveScope 用新建的作用域 标注规则节点，这样下一阶段可以轻易获取作用域。除了处理作用域和函数定义外，我们还需要完成对参数和变量的定义。

## 4.2 解析阶段

下面编写解析阶段的代码，首先将当前作用域设置为定义阶段得到的全局作用域。之后当遍历器触发 Cymbol 函数和代码块的进入和退出方法，根据定义阶段在树中存储的值，将 currentScope 设为相应的作用域。

```

24     public void enterFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
25         currentScope = scopes.get(ctx);
26     }
27     public void exitFunctionDecl(CymbolParser.FunctionDeclContext ctx) {
28         currentScope = currentScope.getEnclosingScope();
29     }
30
31     public void enterBlock(CymbolParser.BlockContext ctx) {
32         currentScope = scopes.get(ctx);
33     }
34     public void exitBlock(CymbolParser.BlockContext ctx) {
35         currentScope = currentScope.getEnclosingScope();
36     }

```

在遍历器正确设置作用域之后，我们就可以在变量引用和函数调用的监听器方法中解析符号了。当遍历器遇到一个变量引用时，它调用 `exitVar()`，该方法使用 `resolve()` 方法在当前作用域的符号表中查找该变量名。如果 `resolve` 方法在当前作用域中没有找到相应的符号，它会沿着外围作用域链查找。必要情况下，`resolve` 将会一直向上查找，直至全局作用域为止。如果它没有找到合适的定义，则返回 `null`。此外，若 `resolve` 方法找到的符号是函数而非变量，我们就需要生成一个错误消息。

## 5.测试程序

下面构建过程和测试过程能够产生之前预期得输出结果。

```
locals:[]
function<f:tINT>:[<x:tINT>, <y:tFLOAT>]
locals:[x, y]
function<g:tVOID>:[]
globals:[f, g]
line 3:4 no such variable: i
line 4:4 g is not a variable
line 13:4 no such function: z
line 14:4 y is not a function
line 15:8 f is not a variable
D:\2022\大一\项目\测试\code\listeners>java -cp D:\antlr-4.9.2-complete.jar;D:\Prof_APPS\ANTLR\antlr-4.9.2-complete.jar;D:\Java\jdk1.8.0_102\lib;D:\Prof_APPS\ANTLR;D:\Prof_APPS\ANTLR\grun.bat;D:\Prof_APPS\ANTLR\antlr4.bat;org.antlr.v4.Tool Cymbol.g4
```