任务一会议纪要

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。

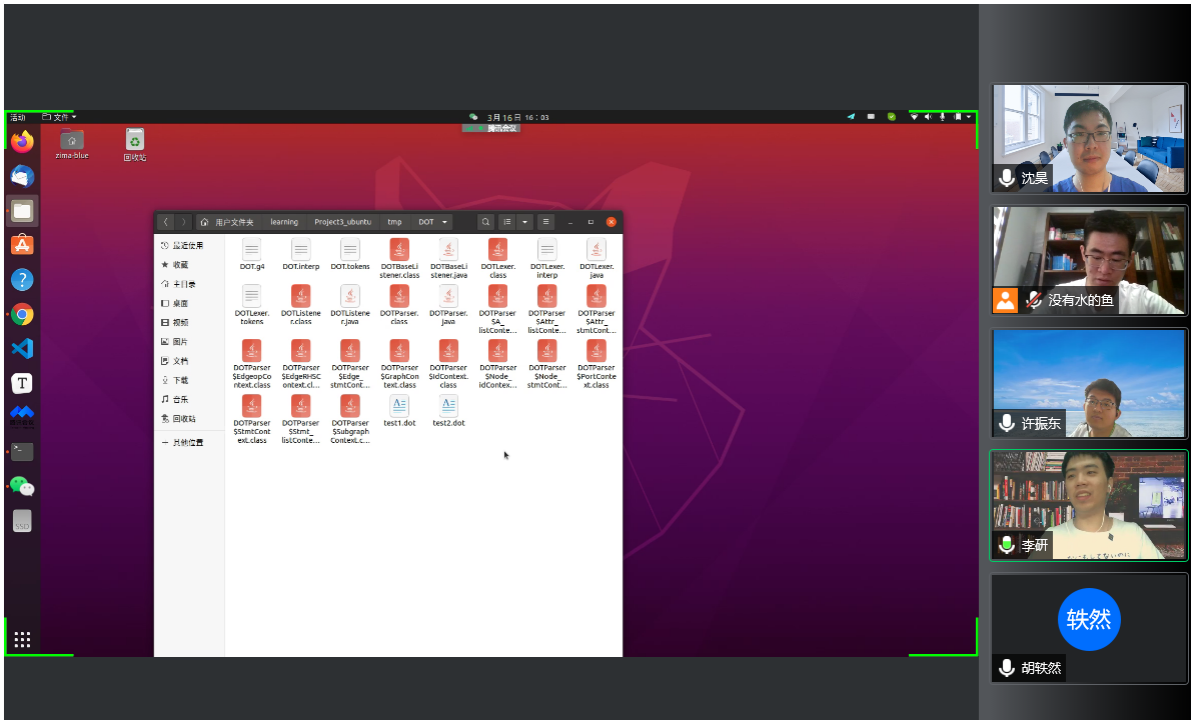
会议纪要记录人：胡轶然

会议纪要审核人：胡书豪

会议记录汇总

1. 任务一的第1次会议：2022.3.10 19：00-22：30 项目制3第1次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），精读至英文参考资料《The Definitive ANTLR4 Reference》的54页，进行了实例的操作和架构的学习。
2. 任务一的第2次会议：2022.3.13 9：00-12：00 项目制3第2次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）精读至英文参考资料《The Definitive ANTLR4 Reference》的102页。完成了csv、json测试用例的编写设计，DOT、symbol和R的测试用例同步进行，预计在下次集体会议期间完成任务一的全部内容。
3. 任务一的第三次会议：2022.3.16 19：00-22：00 项目制3第3次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）完成任务一的全部收尾工作，包括DOT、symbol和R的测试用例的编写，设计文档的书写和会议记录的完善。进一步开启任务二，完成了第七章的学习。关于任务二的学习内容将会在任务二的会议纪要中详细进行介绍。至此，任务一全部完成。

会议拍照：



详细分工

胡轶然：组织开展集体会议；完成Cymbol测试用例的设计；提供会议记录内容并整理书写任务一会议纪要，整理书写任务一设计文档；23%

许振东：完成CSV测试用例的设计；提供会议记录内容；19%

沈昊：完成JSON测试用例的设计；提供会议记录内容；复核测试用例设计文档；20%

李研：完成R测试用例的设计；提供会议记录内容；19%

胡书豪：完成DOT测试用例的设计，提供会议记录内容；复核会议纪要文档；19%

安装

任务一的第1次会议：2022.3.10 19：00-22：30 项目制3第1次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），精读至英文参考资料《The Definitive ANTLR4 Reference》的54页，进行了实例的操作和架构的学习。

在安装前，必须安装好java 1.6及以上的版本。本机的版本为

```
(base) zima-blue@1:/usr/local/lib$ java -version
openjdk version "1.8.0_312"
OpenJDK Runtime Environment (build 1.8.0_312-8u312-b07-0ubuntu1~20.04-b07)
OpenJDK 64-Bit Server VM (build 25.312-b07, mixed mode)
```

```
cd /usr/local/lib/
sudo curl -O https://www.antlr.org/download/antlr-4.7.1-complete.jar #不然会显示没有权限
java -jar antlr-4.7.1-complete.jar  # 启动org.antlr.v4.Tool #测试ANTLR工具是否工作正常
```

因为命令较长，编写脚本运行ANTLR工具的脚本，文件名为antlr：

```
cd ~/learning/Project3 #项目制3工作目录
touch antlr
gedit antlr
```

脚本内容：(把脚本放到usr/local/bin下面就可以全局执行)

```
#!/bin/sh
java -cp /usr/local/lib/antlr-4.7.1-complete.jar org.antlr.v4.Tool $*
```

修改bashrc:`export CLASSPATH=".:/usr/local/lib/antlr-4.0-complete.jar:$CLASSPATH"`(**这里不确定是不是这么加**)。

按照书中的说法，出现了一些小问题：

```
java -jar /usr/local/lib/antlr-4.7.1-complete.jar #可以跑通
java org.antlr.v4.Tool #会报错，错误: 找不到或无法加载主类 org.antlr.v4.Tool
```

修改命令：

```
alias antlr4='java -jar /usr/local/lib/antlr-4.7.1-complete.jar' #每次窗口打开之后都得再次输入,太麻烦，因此把脚本放到usr/local/bin下面。
```

直接把脚本放到usr/local/bin下面就可以全局执行antlr4。已经实现。

```
(base) zima-blue@1:~/learning/Project3/install$ antlr4
ANTLR Parser Generator  Version 4.7.1
 -o ___                specify output directory where all output is generated
 -lib ___              specify location of grammars, tokens files
 -atn                  generate rule augmented transition network diagrams
 -encoding ___         specify grammar file encoding; e.g., euc-jp
 -message-format ___   specify output style for messages in antlr, gnu, vs2005
 -long-messages        show exception details when available for errors and warnings
 -listener             generate parse tree listener (default)
 -no-listener          don't generate parse tree listener
 -visitor              generate parse tree visitor
 -no-visitor           don't generate parse tree visitor (default)
 -package ___          specify a package/namespace for the generated code
 -depend               generate file dependencies
 -D<option>=value      set/override a grammar-level option
 -Werror               treat warnings as errors
 -XdbgST               launch StringTemplate visualizer on generated code
 -XdbgSTWait           wait for STViz to close before continuing
 -Xforce-atn           use the ATN simulator for all predictions
 -Xlog                 dump lots of logging info to antlr-timestamp.log
 -Xexact-output-dir    all output goes into -o dir regardless of paths/package
```

实例操作与架构学习

新建Hello.g4文件：

```
##grammar Hello;
r : 'hello' ID ;
ID : [a-z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

终端执行：

```
antrl4 Hello.g4
ls
javac *.java #报错显示没有javac。重装java和javac,并重启后解决问题
```

进一步：

```
alias grun='java org.antlr.v4.runtime.misc.TestRig'
#这句话只在当前终端有效
grun Hello r -tokens
hello parrt #要在终端输入hello parrt
Ctrl+D on ubuntu
```

显示如下：

```
(base) zima-blue@1:~/learning/Project3/tmp/test$ grun Hello r -tokens
Warning: TestRig moved to org.antlr.v4.gui.TestRig; calling automatically
hello parrt
[@0,0:4='hello',<'hello'>,1:0]
[@1,6:10='parrt',<ID>,1:6]
[@2,12:11='<EOF>',<EOF>,2:0]
```

解释如下：For example, [@1,6:10='parrt',<2>,1:6] indicates that the token is the second token (indexed from 0), goes from character position 6 to 10 (inclusive starting from 0), has text parrt , has token type 2 (ID), is on line 1(from 1), and is at character position 6 (starting from zero and counting tabs as a single character).

进一步在终端打印编译树：

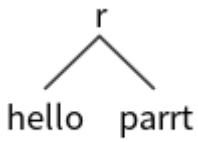
```
grun Hello r -tree
hello parrt #主动输入
Ctrl + D
```

显示如下：

```
(base) zima-blue@1:~/learning/Project3/tmp/test$ grun Hello r -tree
Warning: TestRig moved to org.antlr.v4.gui.TestRig; calling automatically
hello parrt
(r hello parrt)
```

可视化树结构：

```
(base) zima-blue@1:~/learning/Project3/tmp/test$ grun Hello r -gui
Warning: TestRig moved to org.antlr.v4.gui.TestRig; calling automatically
hello parrt
3月 10, 2022 9:58:45 下午 java.util.prefs.FileSystemPreferences$1 run
INFO: Created user preferences directory.
```



在对语法分析树的层次结构、节点类名进行分析后，进一步探究**监视器**和**访问器**的作用和运行流程。

进一步学习了正则表达式的学习和实现、antlr的语法语法概念和实践，例如语法定义文件的通用结构：

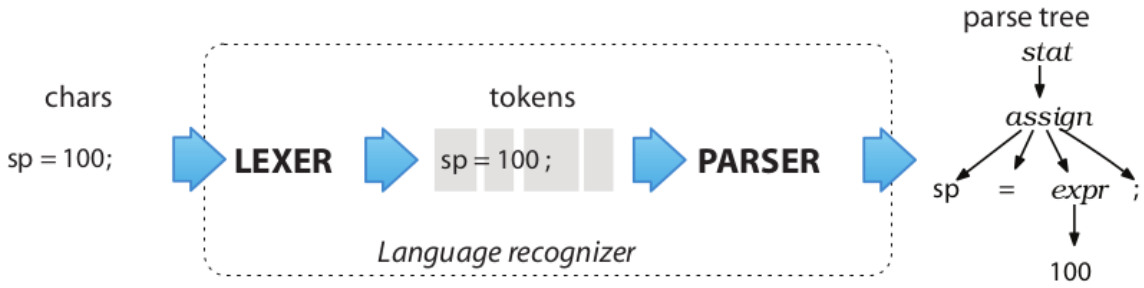
```
/** 可根据需要撰写 javadoc 风格的注释，可以是单行、多行*/
grammar Name;
//注意以下options imports tokens actions指定顺序可以任意调换
 options {name1=value1; ... nameN=valueN;}
import ... ;
tokens { Token1, ..., TokenN }
channels {...} // 只能是词法分析时才能定义
@actionName {...}
rule1 // 语法和词法分析规则定义，也有可能是混合在一起的规则定义
...
ruleN
```

以及antlr生成自顶向下的语法分析器的大致流程和实现框架、antlr解决编程语言中歧义问题的方法实现。

任务一的第2次会议：2022.3.13 9：00-12：00 项目制3第2次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）精读至英文参考资料《The Definitive ANTLR4 Reference》的102页。完成了csv、json测试用例的编写设计，DOT、symbol和R的测试用例同步进行，预计在下次集体会议期间完成任务一的全部内容。

首先主要学习了相关理论知识：

1.ANTLR的整体工作概念流程



2.Parser工作原理和部分判断机制

```
void stat() {
switch ( « current input token » ) {
CASE ID
: assign(); break;
CASE IF
: ifstat(); break; // IF is token type for keyword 'if'
CASE WHILE : whilestat(); break;
...
default
: « raise no viable alternative exception »
    }
}
```

3.歧义的分类和处理：

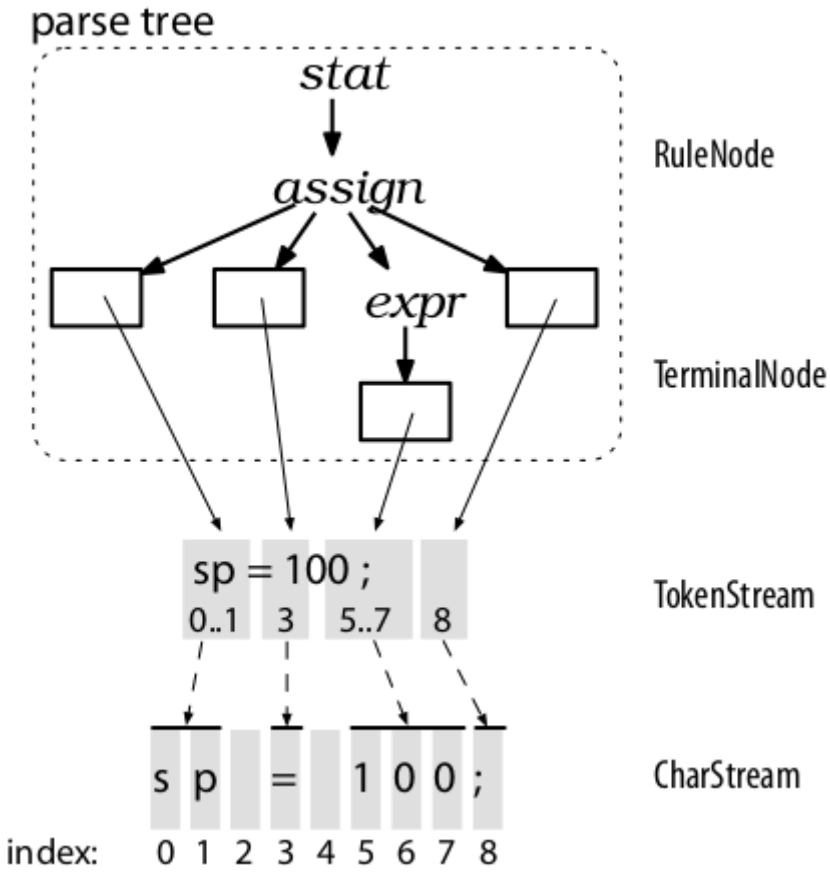
```
//单纯的重复
stat: ID '=' expr ';'
| ID '=' expr ';'
;
expr: INT ;

//some subtle ambiguities
stat: expr:';' // expression statement
| ID '(' ')' ';' // function call statement
;
expr: ID '(' ')'';'
| INT
;
```

3.1. ANTLR resolves the ambiguity by choosing the first alternative involved in the decision.

3.2. ANTLR resolves lexical ambiguities by matching the input string to the rule specified first in the grammar;

```
BEGIN : 'begin' ;
ID: [a-z]+ ;
```

4.lexers工作原理，walk and listen:



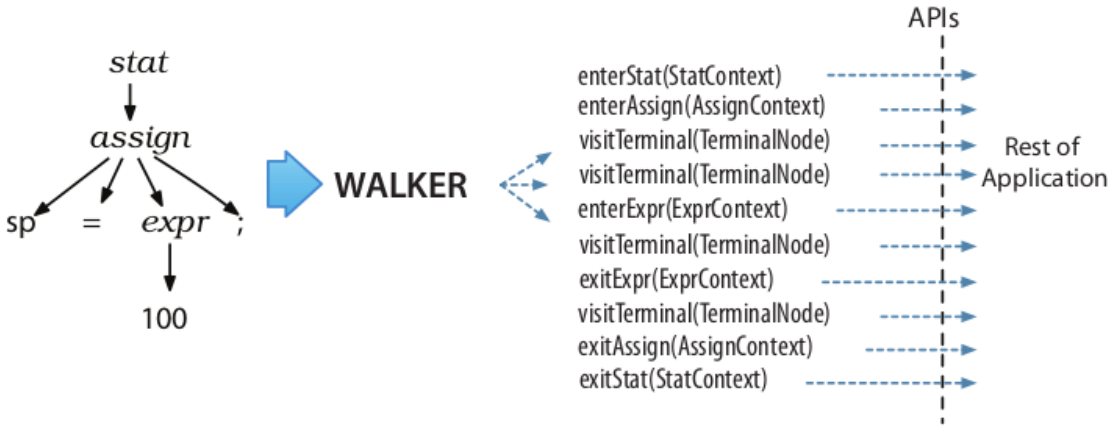


Figure 1—ParseTreeWalker call sequence

之后进入第六章测试用例的编写部分。

测试用例编写

分工如下：

许振东：CSV测试用例

沈昊：Json测试用例

胡轶然：Cymbol测试用例

胡书豪：DOT测试用例

李研：R测试用例

以下内容的详细解释在《任务一测试用例设计文档》可进行阅读查看。

CSV测试用例

CSV语法文件如下：

```
grammar CSV;

file : hdr row+;
hdr : row;
row : field ( ',' field)* '\r' ? '\n' ;
field : TEXT
      | STRING
      |
      ;
TEXT : ~[,\r\n"]+;
STRING : '"' ('""'|~'"')* '"';
```

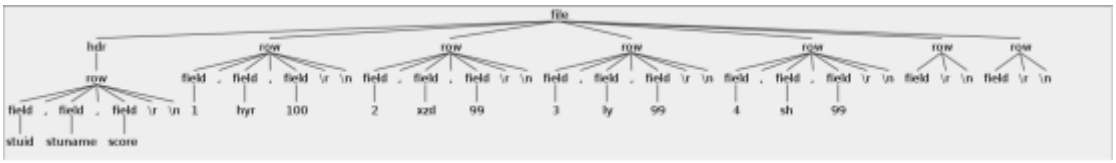
对于第一个测试用例(test1.csv)，简单生成学生成绩表进行尝试，验证语法的可行性：

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

终端输入

```
./grun CSV file -gui test1.csv
```

结果如下：



为了尽可能利用用CSV的所有语法规则，第二个测试用例test2.csv如下：

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

终端输入

```
./grun CSV file -tree test2.csv
```

结果如下：

```
xzd@ubuntu:~$ ./grun CSV file -tree test2.csv
(file (hdr (row (field name) , (field address) , (field goods) \r \n)) (row (fie
ld hyr) , (field "tju") , (field soap) \r \n) (row (field xzd) , (field "tianjin
""wuhu""") , (field chiken) \r \n) (row (field liyan) , (field "daqing") , (fiel
d coke) \r \n) (row field \r \n))
```

Json测试用例

在详细学习第五章语法规则（标点符号、关键字、标识符等）、词法，与语言模式（序列、选择、词法符号依赖、嵌套结构）后，根据任务要求实现了JSON的解析，完成了两个测试用例。

JSON语法文件JSON.g4如下：

```
grammar JSON;

json:  object
    |  array
    ;
object
    :   '{'  pair (',' pair)* '}'
    |   '{'  '}'
    ;
pair:   STRING  ':' value;
array
    :   '[' value (',' value)* ']'
    |   '[' ']'
    ;
value
```

```
    :   STRING
    |   NUMBER
    |   object
    |   array
    |   'true'
    |   'false'
    |   'null'
    ;
STRING :  '"' (ESC | ~["\\])* '"' ;
fragment ESC :  '\\' (["\\/bfnrt] | UNICODE) ;
fragment UNICODE :  'u' HEX HEX HEX HEX ;
fragment HEX :  [0-9a-fA-F] ;
NUMBER
    :   '-'? INT '.' INT EXP?     // 1.35, 1.35E-9, 0.3, -4.5
    |   '-'? INT EXP              // 1e10 -e4
    |   '-'? INT                  // -3, 45
    ;
fragment INT :  '0' | [1-9] [0-9]* ;
fragment EXP :  [Ee] [+\-]? INT ;
WS  :  [ \t\n\r]+ -> skip ;
```

第一个测试用例test0.json中，实现了简单的三元组：

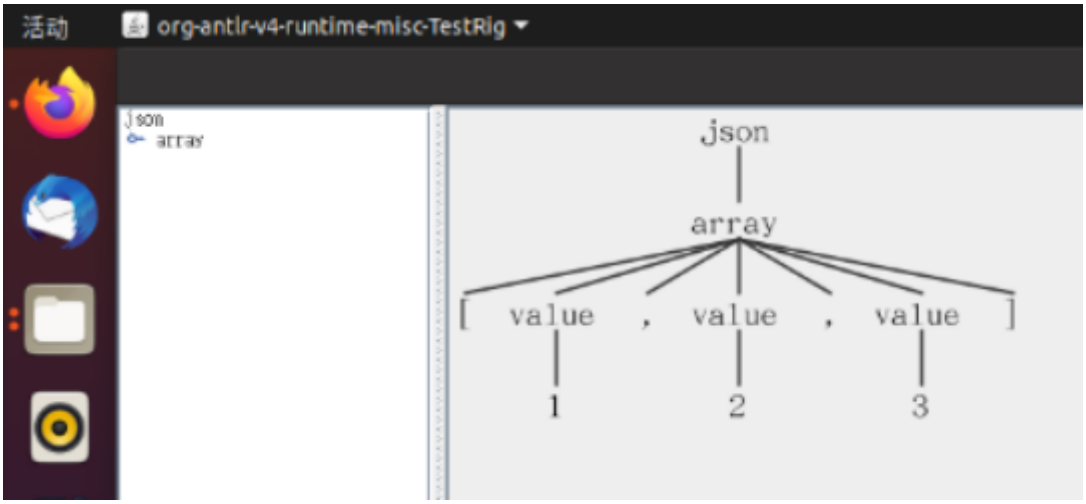
```
[1,2,3]
```

终端输入：

```
alias grun='java org.antlr.v4.runtime.misc.TestRig'
antlr4 JSON.g4
javac JSON*.java
grun JSON json -tree test0.json
grun JSON json -gui test0.json
```

结果如下：

```
uperemehornor@superemehornor-Inspiron-7590:~/ANTLR$ grun JSON json -tree test0.
son
arning: TestRig moved to org.antlr.v4.gui.TestRig; calling automatically
json (array [ (value 1) , (value 2) , (value 3) ]))
uperemehornor@superemehornor-Inspiron-7590:~/ANTLR$
```

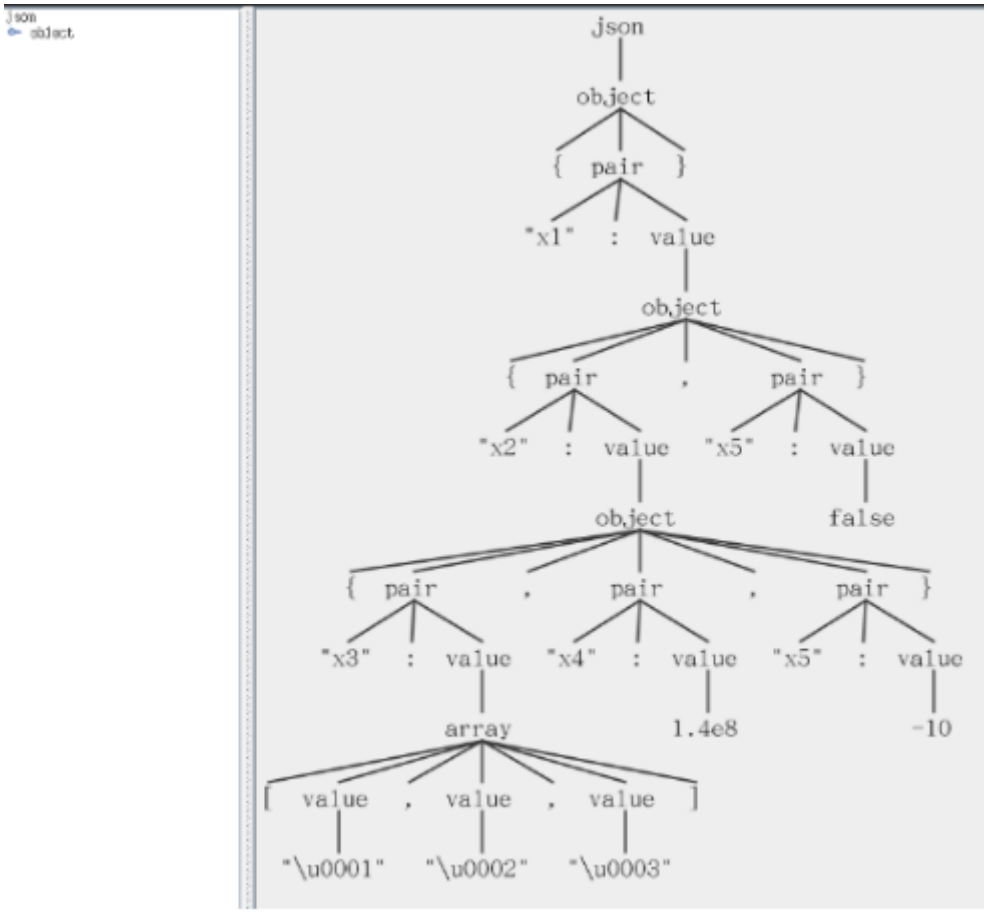


进一步第二个测试用例test1.json中，综合使用了JSON的所有词法与语法，包含了无序键值对集合成的对象、分隔键值对序列所用的逗号、多个数组、循环嵌套、递归调用、各类值（整数、指数型数、Unicode字符序列、字符串）及它们的范围与表达形式、引号的使用、空白字符等。

```
//test1.json
{
      "x1":  {
          "x2" : {
             "x3" : ["\u0001", "\u0002", "\u0003"],
             "x4" : 1.4e8,
             "x5" : -10
            },
          "x5" : false
      }
}
```

结果如下：

```
superemehornor@superemehornor-Inspiron-7590:~/ANTLR$ grun JSON json -tree test1.
json
Warning: TestRig moved to org.antlr.v4.gui.TestRig; calling automatically
(json (object { (pair "x1" : (value (object { (pair "x2" : (value (object { (pai
r "x3" : (value (array [ (value "\u0001") , (value "\u0002") , (value "\u0003")
])) , (pair "x4" : (value 1.4e8)) , (pair "x5" : (value -10)) })) , (pair "x5"
 : (value false)) })) }))
superemehornor@superemehornor-Inspiron-7590:~/ANTLR$
```



Cymbol测试用例

任务一的第三次会议：2022.3.16 19：00-22：00 项目制3第3次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）完成任务一的全部收尾工作，包括DOT、symbol和R的测试用例的编写，设计文档的书写和会议记录的完善。进一步开启任务二，完成了第七章的学习。关于任务二的学习内容将会在任务二的会议纪要中详细进行介绍。至此，任务一全部完成。

首先，按照书中的语法介绍，构建语法文件Cymbol.g4:

```
grammar Cymbol;

file //function declarations and variable declarations
    : (functionDecl | varDecl)+
    ;

varDecl // variable declarations, such as int x = 5;
    : type ID ('=' expr)? ';';
type: 'float' | 'int' | 'void';// user-defined types

functionDecl // function declarations, such as "void f(int x) {...}"
    : type ID '(' formalParameters? ')' block
    ;
formalParameters // int x, float y
    : formalParameter (',' formalParameter)*
    ;
formalParameter // int x
    : type ID
    ;

//the gramma of function body:由花括号包围的语句块。 有六种语句：嵌套块、变量声明、if 语句、return 语句、赋值和函数调用。
block
    : '{' stat* '}'
    ;
stat
    : block
    | varDecl
    | 'if' expr 'then' stat ('else' stat)?
    | 'return' expr? ';'
    | expr '=' expr ';' // assignment
    | expr ';' // function call
    ;

expr //最后一个主要的部分是表达式语法，包括：一元否定、布尔非、乘法、加法、减法、函数调用、数组索引、相等比较、变量、整数和括号表达式。
    : ID '(' exprList? ')' // functional call, f(), f(x), f(x, 2)
    | expr '[' expr ']'    // array index, a[i], a[2], a[1][i]
    | '-' expr             // unary minus
    | '!' expr             // boolean not
    | expr '*' expr        // multiplication
    | expr ('+'|'-') expr  // addition or subtraction
    | expr '==' expr       // equalit comparison, lowest priority operator
    | ID                   // variable reference
    | INT
    | '(' expr ')'
    ;
```

```
exprList
    : expr (',' expr)* ;   // arg list

ID  :   LETTER (LETTER | [0-9])* ;
fragment
LETTER : [a-zA-Z] ;

INT :   [0-9]+ ;

WS  :   [ \t\n\r]+ -> skip ;

SL_COMMENT
    :   '//' .*? '\n' -> skip
    ;
```

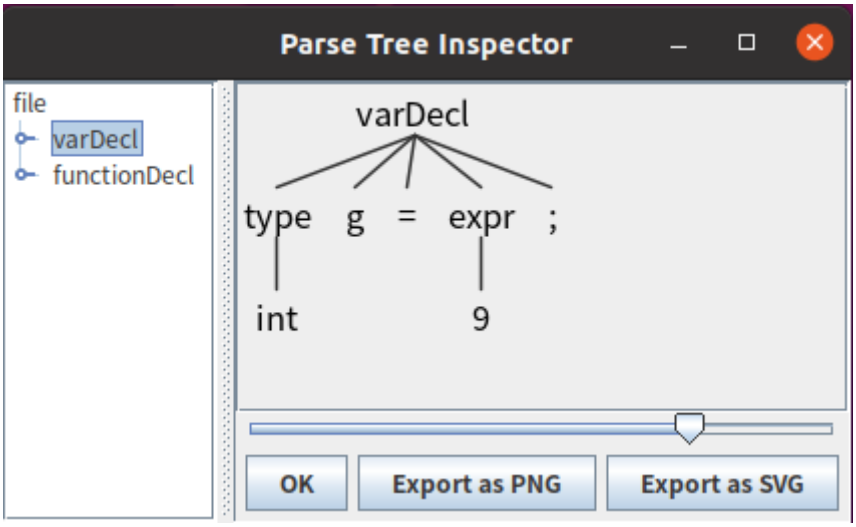
先运行书中的测试用例，t.cymbol:

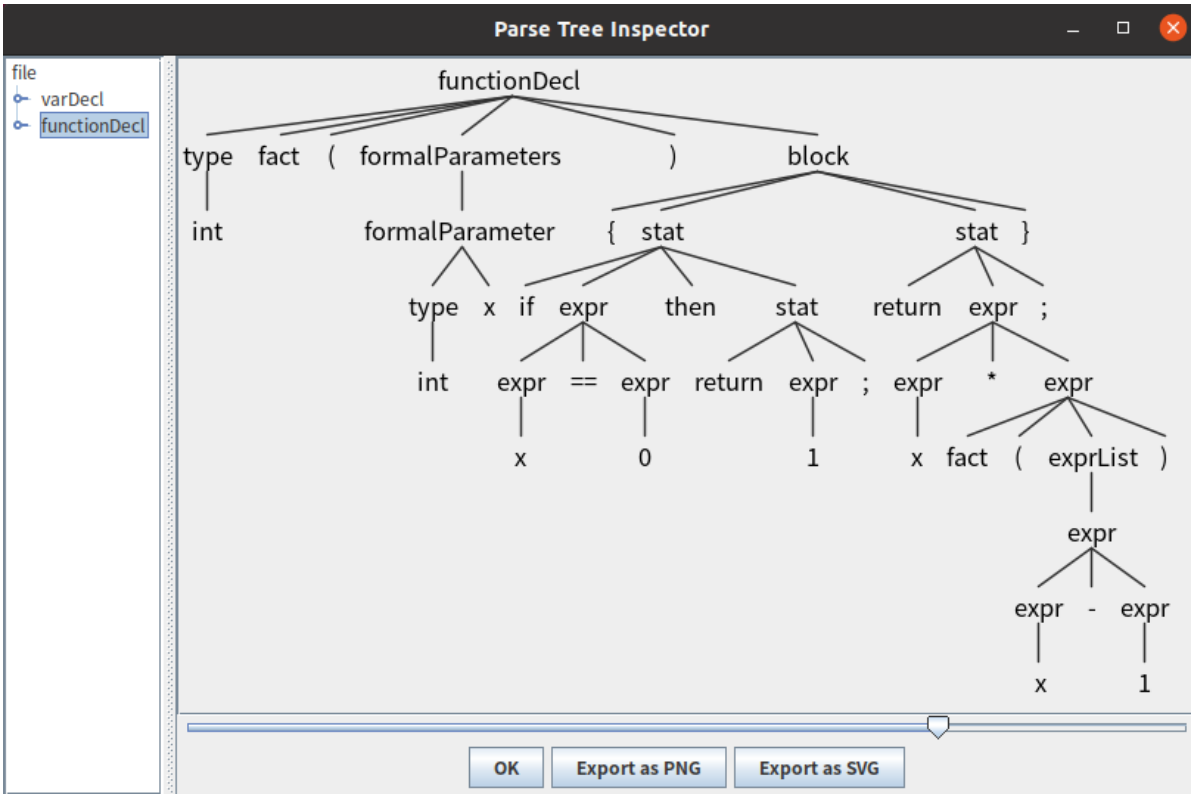
```
// Cymbol test
int g = 9; // a global variable
int fact(int x) { // factorial function
if x==0 then return 1;
return x * fact(x-1);
}
```

终端输入如下命令：

```
alias grun='java org.antlr.v4.runtime.misc.TestRig'
antlr4 Cymbol.g4
javac Cymbol*.java
grun Cymbol file -gui t.cymbol
```

生成的解析树如下：





该书本样例t.cymbol体现的Cymbol语法规则如下：

```
1.function declarations and variable declarations
2.if-else structure
3.the gramma of function body(if-else, return, assignment)
4.functional call
5.equalit comparison and subtraction
```

表达式语法中的：一元否定、布尔非、乘法、加法、数组索引、括号表达式没有涉及。为了全面测试cymbol的语法，编写如下测试用例，全面覆盖symbol的语法（发现按照既定的语法文件，cymbol是不支持循环语句的）。

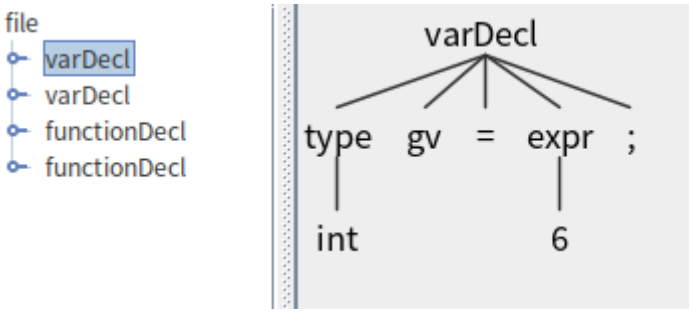
test1.cymbol:

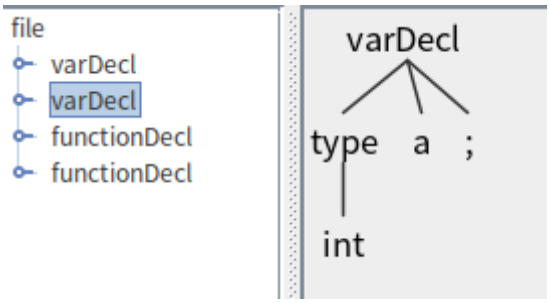
```
// Cymbol test1 by hyr
int gv = 6; // 全局变量声明
int a;
int plus(int a){
    return x+1; //加法
}

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);//函数调用
}
```

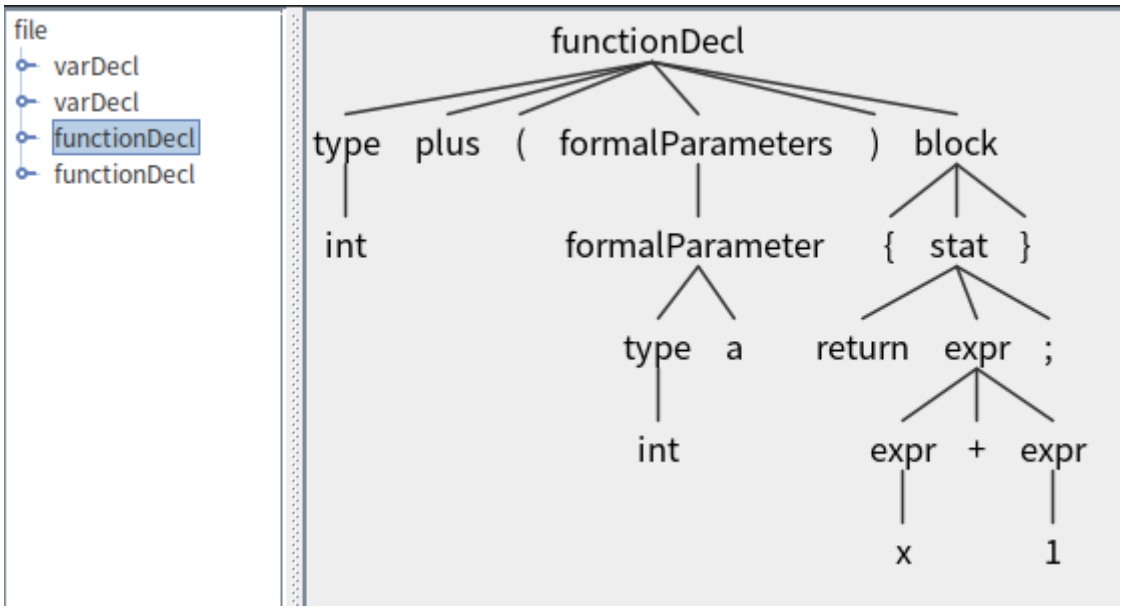
终端输入：

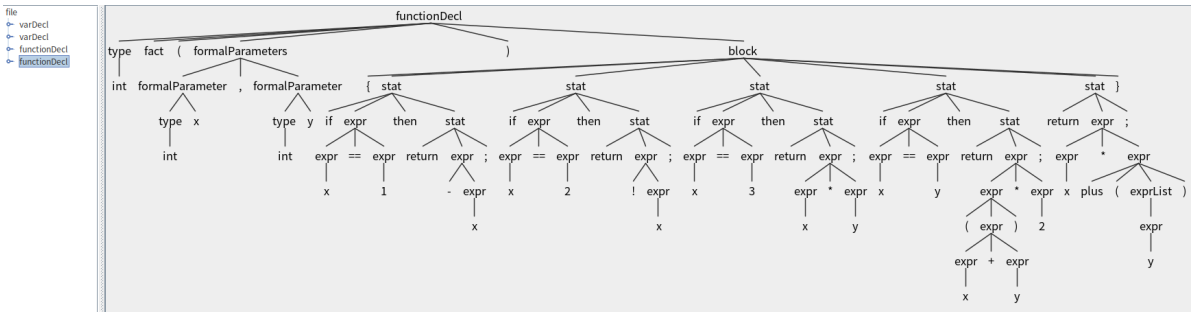
```
alias grun='java org.antlr.v4.runtime.misc.TestRig'
antlr4 Cymbol.g4
javac Cymbol*.java
grun Cymbol file -gui test1.cymbol
```

结果如下：









DOT测试用例

DOT的语法文件如下：

```
grammar DOT;

graph
    : STRICT? (GRAPH | DIGRAPH) id? '{' stmt_list '}' ;
stmt_list
    : ( stmt ';' ? )* ;
stmt
    : node_stmt
    | edge_stmt
    | attr_stmt
    | id '=' id
    | subgraph
    ;
attr_stmt
    : (GRAPH | NODE | EDGE) attr_list ;
attr_list
    : ('[' a_list? ']')+ ;
a_list
    : (id ('=' id)? ','?)+ ;
edge_stmt
    : (node_id | subgraph) edgeRHS attr_list? ;
edgeRHS
    : (edgeop (node_id | subgraph) )+ ;
edgeop
    : '->' | '--' ;
node_stmt
    : node_id attr_list? ;
node_id
    : id port? ;
port
    : ':' id (':' id)? ;
subgraph
    : (SUBGRAPH id?)? '{' stmt_list '}' ;
id
    : ID
    | STRING
    | HTML_STRING
    | NUMBER
    ;

STRICT
    : [Ss][Tt][Rr][Ii][Cc][Tt] ;
GRAPH
    : [Gg][Rr][Aa][Pp][Hh] ;
DIGRAPH
    : [Dd][Ii][Gg][Rr][Aa][Pp][Hh] ;
NODE
    : [Nn][Oo][Dd][Ee] ;
EDGE
    : [Ee][Dd][Gg][Ee] ;
SUBGRAPH
    : [Ss][Uu][Bb][Gg][Rr][Aa][Pp][Hh] ;
```

```
ID
    : LETTER (LETTER|DIGIT)* ;
NUMBER
    : '-'? ('.' DIGIT+ | DIGIT+ ('.' DIGIT*)? ) ;
STRING
    : '"' ('\\"'|.)*? '"' ;
HTML_STRING
    : '<' (TAG|~[<>])* '>' ;

fragment LETTER
    : [a-zA-Z\u0080-\u00FF_] ;
fragment DIGIT
    : [0-9] ;
fragment TAG
    : '<' .*? '>' ;

PREPROC
    : '#' .*? '\n' -> skip ;
COMMENT
    :   '/*' .*? '*/' -> skip ;
LINE_COMMENT
    :   '//' .*? '\r'? '\n' -> skip ;
WS
: [ \t\n\r]+ -> skip ;
```

设计的两个测试用例包含了dot语言中所有的关键字和特定字符，基本上实现了dot语言中全部的语法规则形式，如图边节点的全局属性和局部属性的定义、子图及子图的继承与属性覆盖、罗盘端口和命名端口、单行注释和多行注释等。

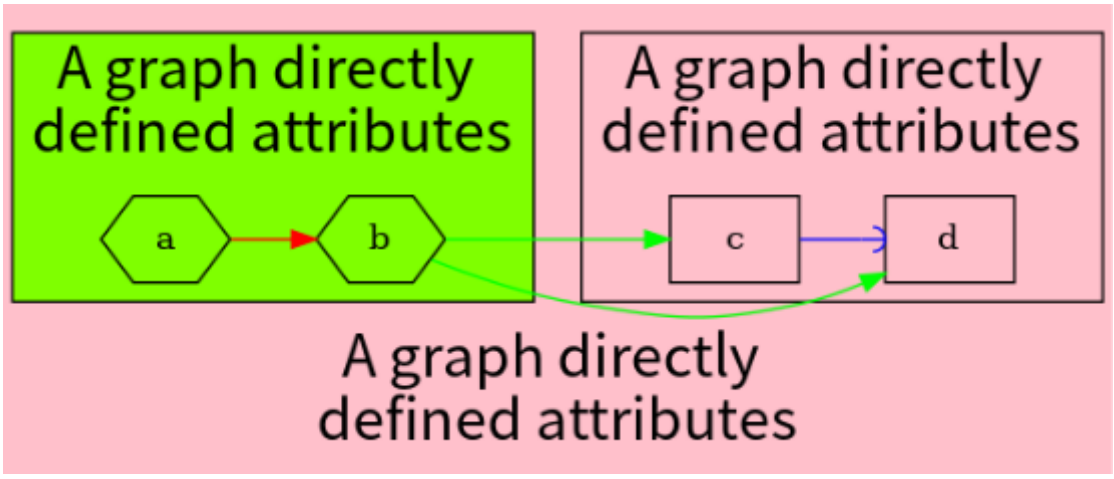
第一个测试用例覆盖了DOT语言中全部的关键字和字符（除了竖杠|，在下个测试用例中进行了测试），并实现了图、节点、连线的常用全局属性和一些局部属性，如颜色、样式、字体、连线箭头图绘制方向等：

test1.dot

```
strict digraph abc{
    label="A graph directly \ndefined attributes";
    bgcolor=skyblue;
    fontname="Microsoft Yahei";
    fontsize=24;
    rankdir=LR;

    graph [bgcolor=pink];
    edge [color=blue];
    node [shape=box];

    subgraph cluster1 {
        graph [bgcolor=chartreuse];
        edge [color=red];
        node [shape=hexagon];
        a -> b;
    }

    subgraph cluster2 {
        c -> d [arrowhead=curve];
    }
```

```
    b -> {c d} [color=green];
    //I like this course
    /* hahaha
    */
}
```

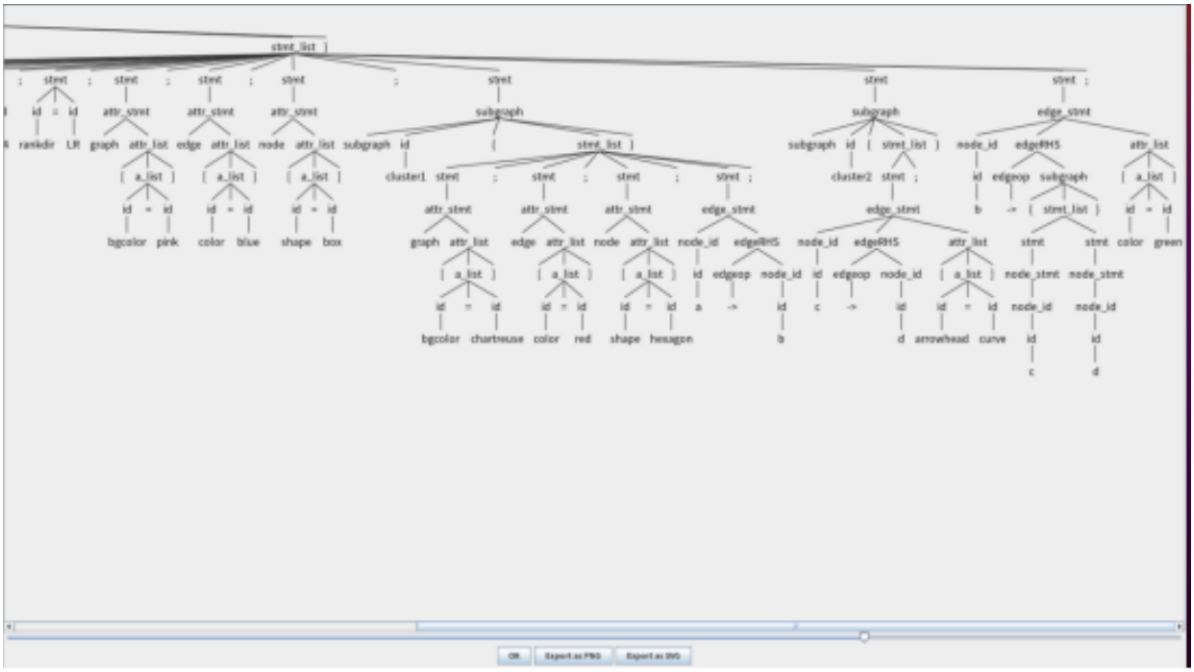
该dot文件生成的图结构如下：



终端输入：

```
./grun DOT graph -gui test1.dot
```

结果如下：

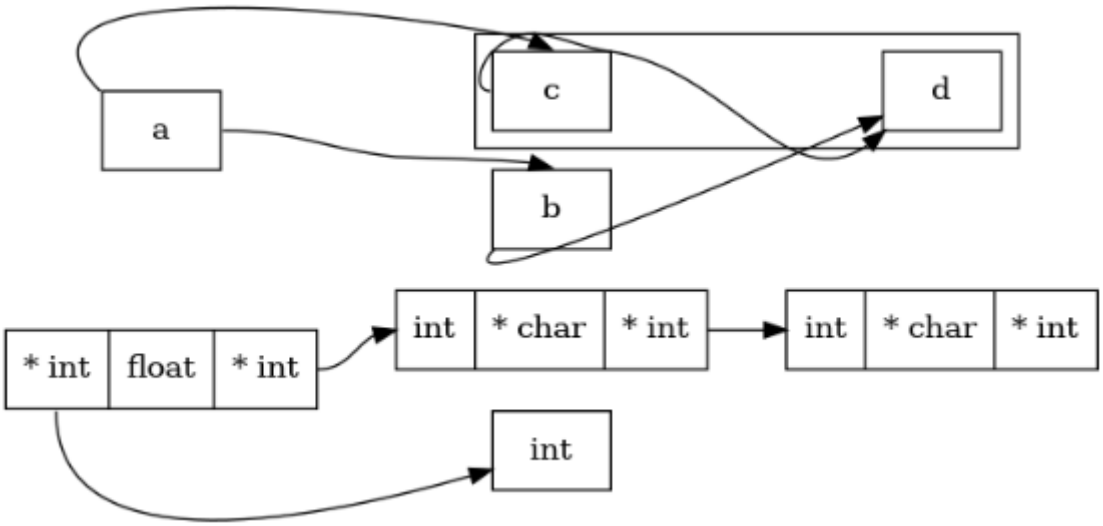


第二个测试用例test.dot使用了DOT语言中部分的关键字和字符（包括竖杠 |），并实现了罗盘端口和命名端口的语法规则形式。

test2.dot

```
digraph {
    rankdir=LR;
    fontsize=12;
    node [shape=record];

    struct1 [label="{<head>* int|float|<next>* int}"];
    struct2 [label="{<prev>int|* char|<next>* int}"];
    struct3 [label="{<prev>int|* char|* int}"];

    struct1:next -> struct2:prev;
    struct2:next -> struct3:prev;
    struct1:head:s -> int;

    node [shape=box];
    a:e -> b:n;
    a:nw -> c:n;
    c:w -> d:sw;
    b:sw -> d;
}
```

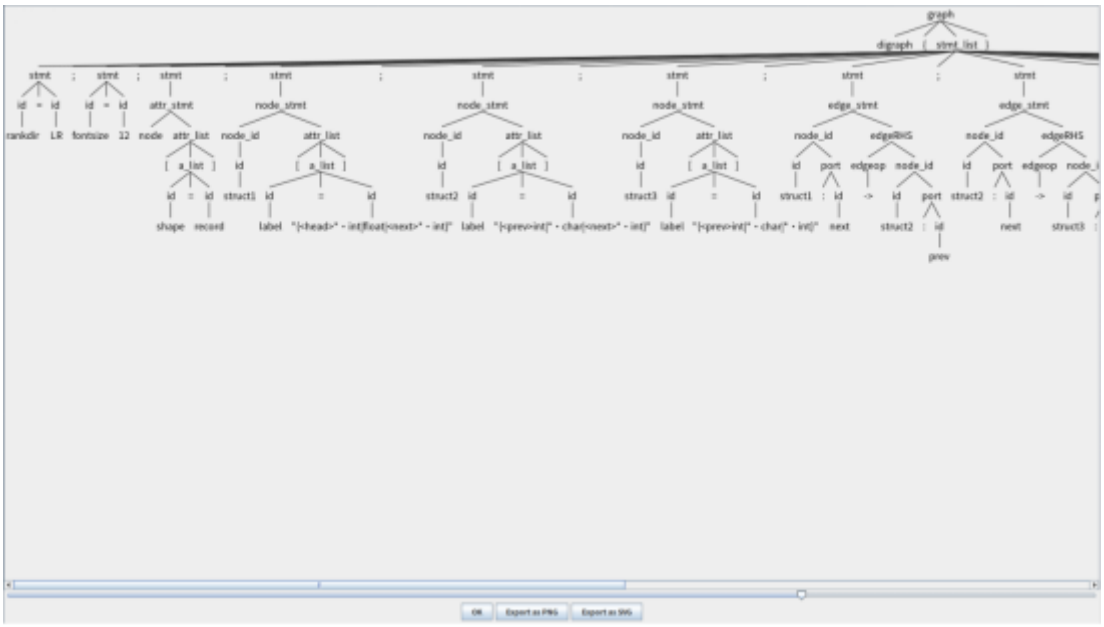
该dot文件生成的图结构如下：



终端输入

```
./grun DOT graph -gui test2.dot
```

结果如下：



R测试用例

R的语法文件如下：

```
grammar R;

prog
    : ( expr_or_assign (';'|NL) | NL)* EOF
    ;

expr_or_assign
    : expr ('<-'|'='|'<<-') expr_or_assign
    | expr
    ;

expr:   expr '['']' sublist ']' ']'  // '[[' follows Rs yacc grammar
    |   expr '[' sublist ']'
    |   expr ('::'|':::') expr
    |   expr ('$'|'@') expr
    |   <assoc=right> expr '^' expr
    |   ('-'|'+') expr
    |   expr ':' expr
    |   expr USER_OP expr // anything wrappedin %: '%' .* '%'
    |   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   expr ('>'|'>='|'<'|'<='|'=='|'!=') expr
    |   '!' expr
    |   expr ('&'|'&&') expr
    |   expr ('|'|'||') expr
    |   '~' expr
    |   expr '~' expr
    |   expr ('->'|'->>'|':=') expr
    |   'function' '(' formlist? ')' expr // define function
    |   expr '(' sublist ')'              // call function
    |   '{' exprlist '}' // compound statement
    |   'if' '(' expr ')' expr 'else' expr
    |   'if' '(' expr ')' expr
    |   'for' '(' ID 'in' expr ')' expr
    |   'while' '(' expr ')' expr
    |   'repeat' expr
    |   '?' expr // get help on expr, usually string or ID
    |   'next'
    |   'break'
    |   '(' expr ')'
    |   ID
    |   STRING
    |   HEX
    |   INT
    |   FLOAT
    |   COMPLEX
    |   'NULL'
    |   'NA'
    |   'Inf'
    |   'NaN'
    |   'TRUE'
```

```
    |   'FALSE'
    ;

exprlist
    : expr_or_assign ((';'|NL) expr_or_assign? )*
    |
    ;

formlist
    : form (',' form)*
    ;

form
    : ID
    | ID '=' expr
    | '...'
    ;

sublist
    : sub (',' sub)* ;

sub
    : expr
    | ID '='
    | ID '=' expr
    | STRING '='
    | STRING '=' expr
    | 'NULL' '='
    | 'NULL' '=' expr
    | '...'
    |
    ;


HEX :   '0' ('x'|'X') HEXDIGIT+ [Ll]? ;

INT :   DIGIT+ [Ll]? ;

fragment
HEXDIGIT : ('0'..'9'|'a'..'f'|'A'..'F') ;

FLOAT:  DIGIT+ '.' DIGIT* EXP? [Ll]?
    |   DIGIT+ EXP? [Ll]?
    |   '.' DIGIT+ EXP? [Ll]?
    ;
fragment
DIGIT:  '0'..'9' ;
fragment
EXP :   ('E' | 'e') ('+' | '-')? INT ;

COMPLEX
    :   INT 'i'
    |   FLOAT 'i'
    ;

STRING
    :   '"' ( ESC | ~[\\"] )*? '"'
    |   '\'' ( ESC | ~[\\'] )*? '\''
```

```
    ;

fragment
ESC :   '\\' ([abtnfrv]|'"'|'\'')
    |   UNICODE_ESCAPE
    |   HEX_ESCAPE
    |   OCTAL_ESCAPE
    ;

fragment
UNICODE_ESCAPE
    :   '\\' 'u' HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT
    |   '\\' 'u' '{' HEXDIGIT HEXDIGIT HEXDIGIT HEXDIGIT '}'
    ;

fragment
OCTAL_ESCAPE
    :   '\\' [0-3] [0-7] [0-7]
    |   '\\' [0-7] [0-7]
    |   '\\' [0-7]
    ;

fragment
HEX_ESCAPE
    :   '\\' HEXDIGIT HEXDIGIT?
    ;

ID  :   '.' (LETTER|'_'|'.') (LETTER|DIGIT|'_'|'.')*
    |   LETTER (LETTER|DIGIT|'_'|'.')*
    ;
fragment LETTER  : [a-zA-Z] ;

USER_OP :   '%' .*? '%' ;

COMMENT :   '#' .*? '\r'? '\n' -> type(NL) ;

// Match both UNIX and Windows newlines
NL      :   '\r'? '\n' ;

WS      :   [ \t]+ -> skip ;
```

为了覆盖R语法中的大部分规则，设计如下测试用例：

test1.R

```
myString <- "Hello, World!"
arbit = c(0,1,2,3)
klasse <- 9+2*4/8-5 <= 8
name <- !(8 >= 4 & 8 <= 9.5)
name1 <- ~(-8 < 7 && +8 > 2)
name2 <- 12 == 12 || 12 != 9
name8 <<- 4 | 5
name11 :: name12
name13 ::: name14
name15 $ name16
name17 @ name18
if (name1 == name2) name4 <- TRUE
```

```
round(1.5)
name5 <- {name1 + name2}
name6 <- 1
name7 <- 1
for ( name6 in 5) name7 <- name7 + 1
nameloop <= 1
while(nameloop < 10) nameloop <- nameloop + 1
repeat break
?name2
print ( myString )
```

对于该测试用例的详细解读如下：

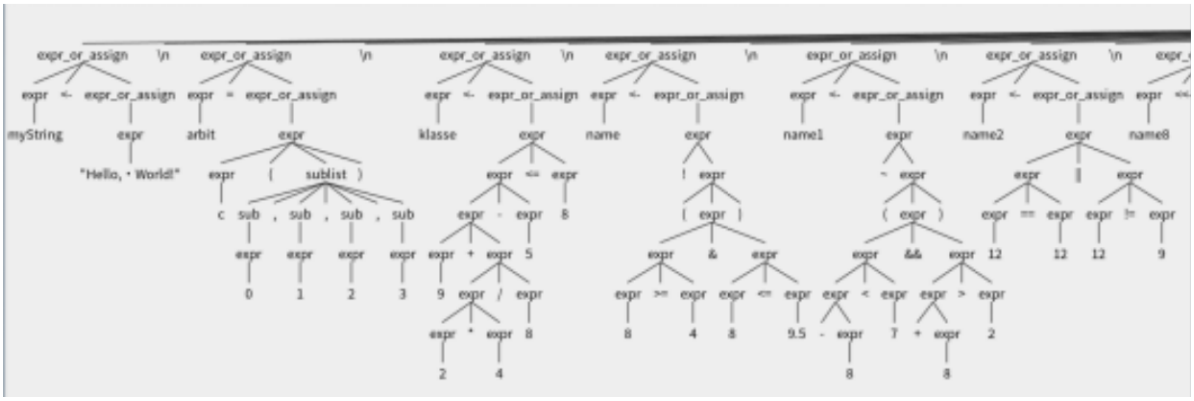
前11行包括R语言支持的大多数运算符和原子表达式（<-、=、<<-、（）、+、-、*、/、<=、>=、<、>、=、=、！=、！、~、&&、&、&、&、|、|、：、：、$、@、，ID、INT、STRING、FLOAT、exprlist、formlist、form、sublist、sub、TRUE和FALSE），但覆盖所有分支似乎是不可能的（例如，覆盖整个“sublist规则”需要18行示例）。

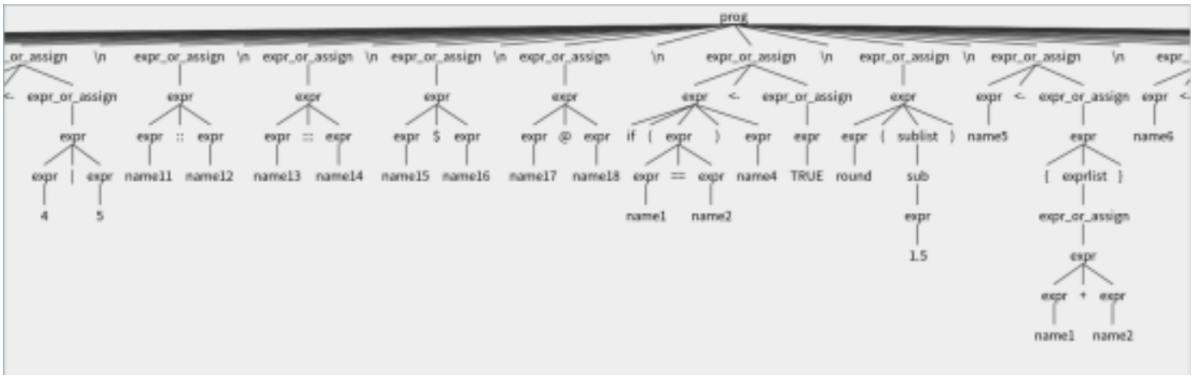
第12~22行包括条件语句的分支、循环以及函数的定义和调用。这些行测试规则是否支持基本循环和函数。

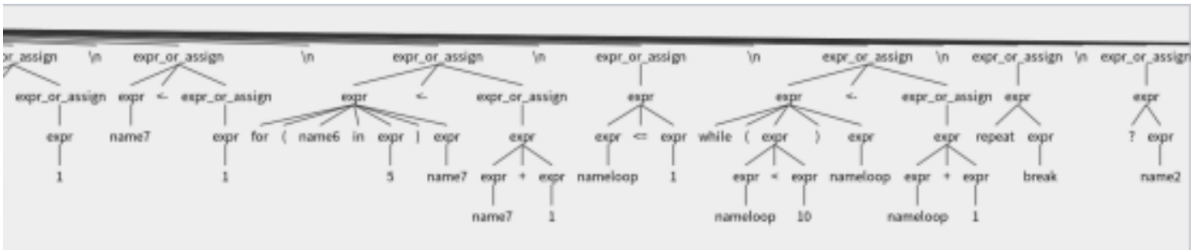
终端输入：

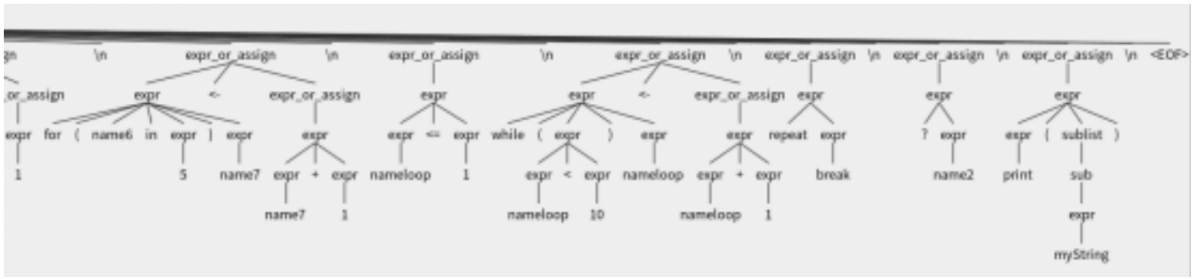
```
antlr4 R.g4
javac R*.java
grun R prog -gui test1.R:
```

由于树的结构较为庞大，树结构从左到右的分布截图如下：









至此，小组完成任务一测试用例的编写。