天津大学

任务一测试用例设计与操作说明

学院 智能与计算学部

班级 计算机科学与技术

组长 胡轶然 3019244355

成员 许振东 3019244356

沈昊 3019234165

李研 3019205150

胡书豪 3019205412

目录

任务概述

法文件根据书中的语法文件和测试用例，分别针对 CSV、JSON、DOT、Cymbol 和 R 设计不同于书中的 1-2 个测试用例，分析设计思路、操作方式并进行语法分析树的呈现。

小组分工

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%

CSV 测试用例设计

CSV 语法文件如下：

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

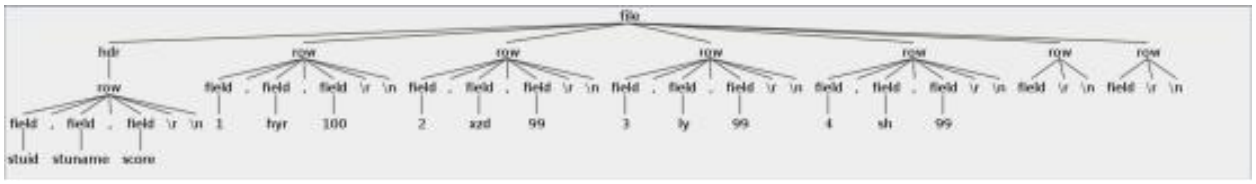
根据语法文件，设计第一个测试用例(test1.csv)，简单生成学生成绩表进行尝试，验证语法的可行性：

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

终端操作步骤：

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

结果如下：



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

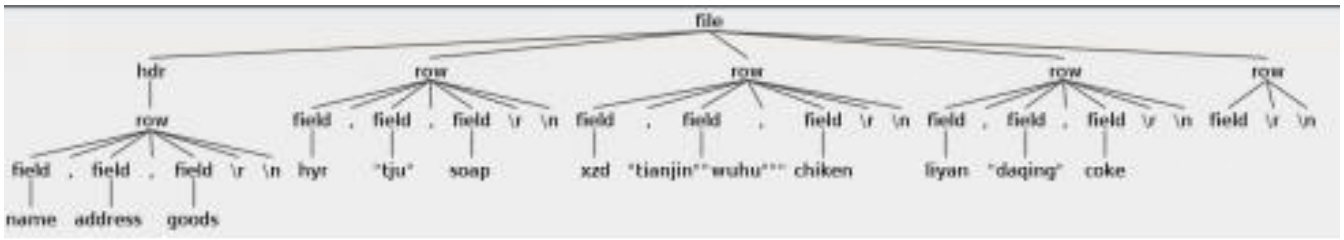
```
1. name,address,goods
2. hyr,"tju",soap
3. xzd,"tianjin""wuhu""",chiken
4. liyan,"daqing",coke
```

终端操作步骤：

1. ./grun CSV file -tree test2.csv
2. ./grun CSV file -gui 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 测试用例设计

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：

```
1. grammar JSON;
```

```
2.
3.  json:  object
4.      | array
5.      ;
6.  object
7.      :  '{'  pair (',' pair)* '}'
8.      |  '{'  '}'
9.      ;
10. pair:   STRING  ':' value;
11. array
12.     :  '[' value (',' value)* ']'
13.     |  '[' ']'
14.     ;
15. value
16.     :  STRING
17.     |  NUMBER
18.     |  object
19.     |  array
20.     |  'true'
21.     |  'false'
22.     |  'null'
23.     ;
24. STRING :  '"' (ESC | ~["\\])* '"' ;
25. fragment ESC :  '\\' (["\\/bfnrt] | UNICODE) ;
26. fragment UNICODE :  'u' HEX HEX HEX HEX ;
27. fragment HEX :  [0-9a-fA-F] ;
28. NUMBER
29.     :  '-'? INT '.' INT EXP?     // 1.35, 1.35E-9, 0.3, -4.5
30.     |  '-'? INT EXP              // 1e10 -e4
31.     |  '-'? INT                  // -3, 45
32.     ;
33. fragment INT :  '0' | [1-9] [0-9]* ;
34. fragment EXP :  [Ee] [+\-]? INT ;
35. WS  :  [ \t\n\r]+ -> skip ;
```

在第一个测试用例 test0.json 中，实现了简单的三元组：

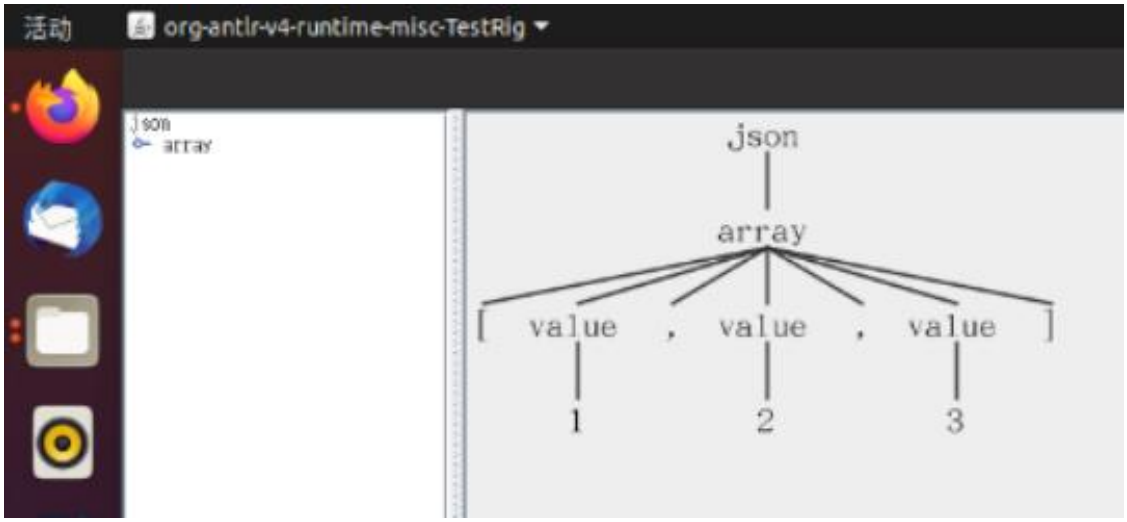
[1,2,3]

终端操作步骤如下：

```
1.  alias grun='java org.antlr.v4.runtime.misc.TestRig'
2.  antlr4 JSON.g4
3.  javac JSON*.java
4.  grun JSON json -tree test0.json
5.  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$
```

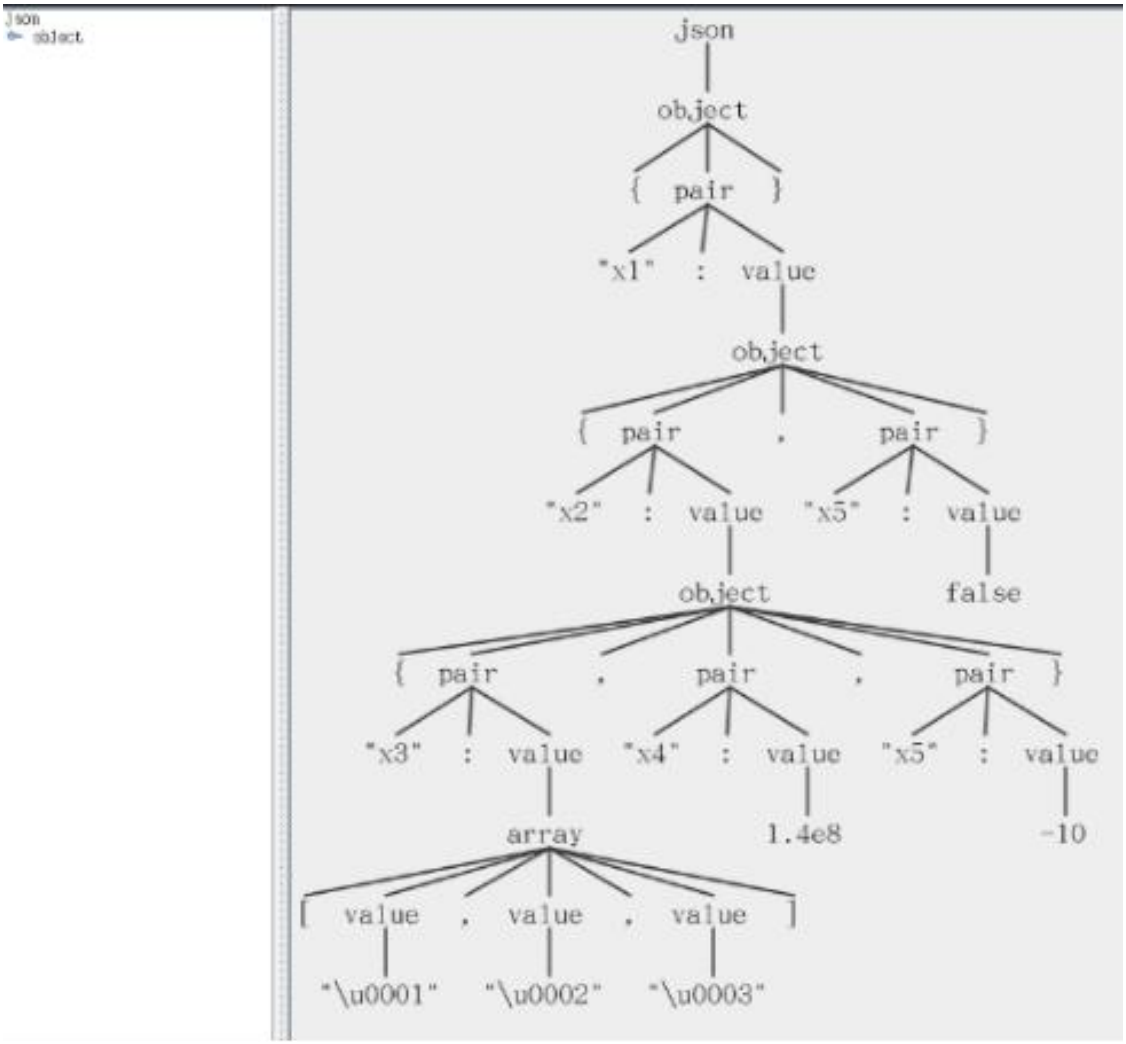


进一步第二个测试用例 test1.json 中，综合使用了 JSON 的所有词法与语法，包含了无序键值对集合成的对象、分隔键值对序列所用的逗号、多个数组、循环嵌套、递归调用、各类值（整数、指数型数、Unicode 字符序列、字符串）及它们的范围与表达形式、引号的使用、空白字符等。

```
1.  //test1.json
2.  {
3.      "x1":  {
4.          "x2" : {
5.              "x3" : ["\u0001", "\u0002", "\u0003"],
6.              "x4" : 1.4e8,
7.              "x5" : -10
8.              },
9.          "x5" : false
10.     }
11. }
```

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 测试用例设计

首先，按照书中的语法介绍，构建语法文件 Cymbol.g4:

```
1.  grammar Cymbol;
2.
3.  file //function declarations and variable declarations
4.      : (functionDecl | varDecl)+
5.      ;
6.
7.  varDecl // variable declarations, such as int x = 5;
8.      : type ID ('=' expr)? ';';
9.  type: 'float' | 'int' | 'void';// user-defined types
10.
11. functionDecl // function declarations, such as "void f(int x) {...}"
12.     : 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 ;
```

```
56.
57. SL_COMMENT
58.     :  '//' .*? '\n' -> skip
59.     ;
```

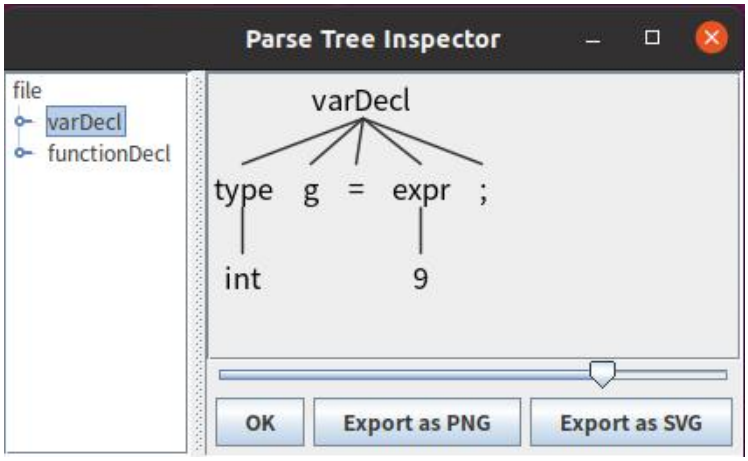
先运行书中的测试用例，t.cymbol:

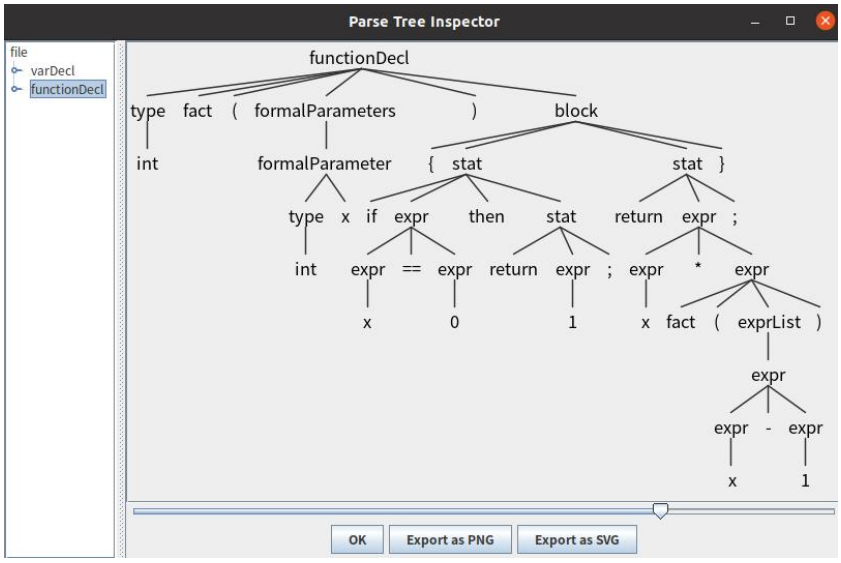
```
1.  // Cymbol test
2.  int g = 9; // a global variable
3.  int fact(int x) { // factorial function
4.  if x==0 then return 1;
5.  return x * fact(x-1);
6.  }
```

终端输入如下命令：

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

生成的解析树如下：





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 test1.cymbol，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）。

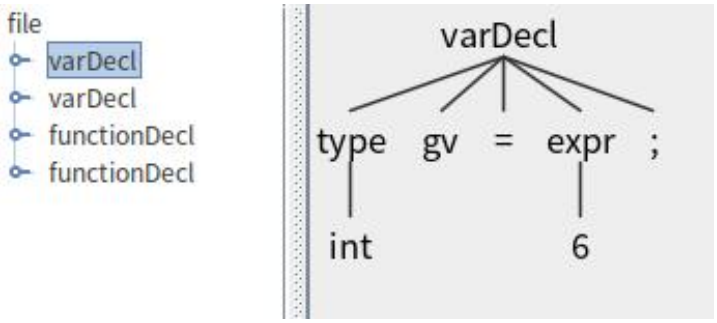
```
1.  // Cymbol test1 by hyr
2.  int gv = 6; // 全局变量声明
3.  int a;
4.  int plus(int a){
5.     return x+1; //加法
6.  }
7.
8.  int fact(int x, int y) { // factorial function
9.  if x==1 then return -x; //一元否定
10. if x==2 then return !x; //布尔非
11. if x==3 then return x*y; //乘法
12. if x==y then return (x+y)*2; //括号表达式
13. return x * plus(y);//函数调用
14. }
```

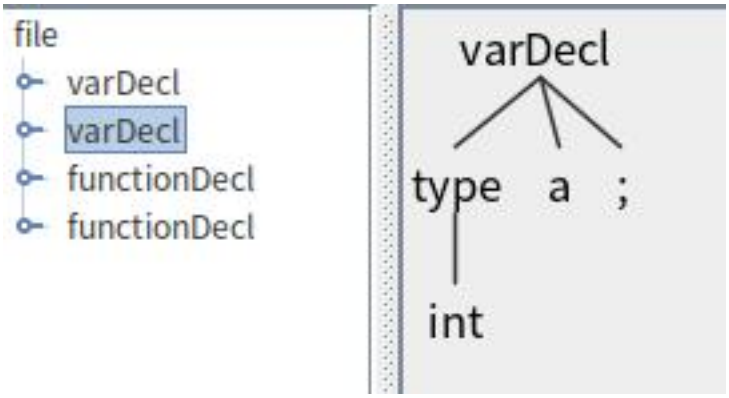
终端操作步骤：

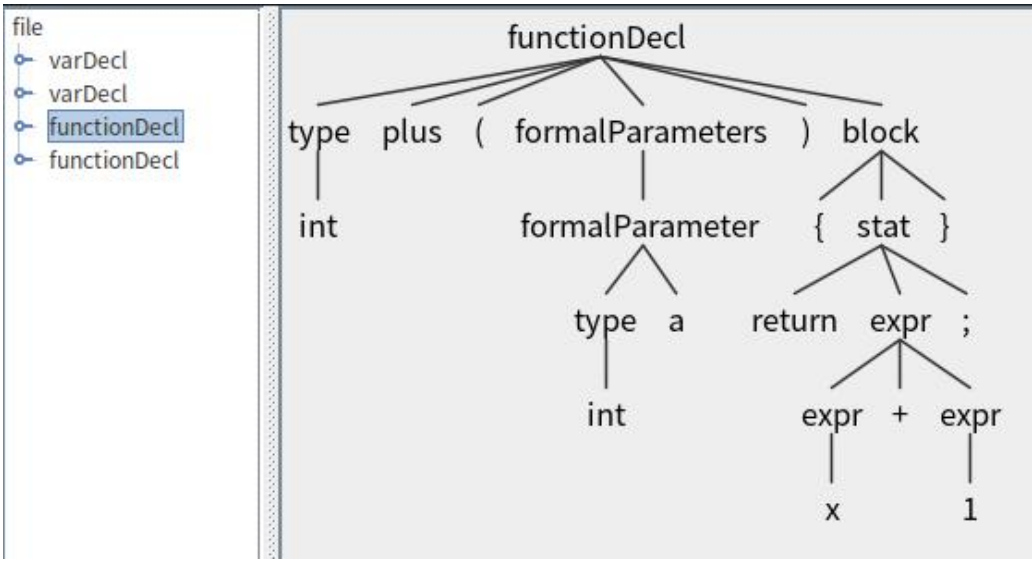
```
1.  alias grun='java org.antlr.v4.runtime.misc.TestRig'
2.  antlr4 Cymbol.g4
```

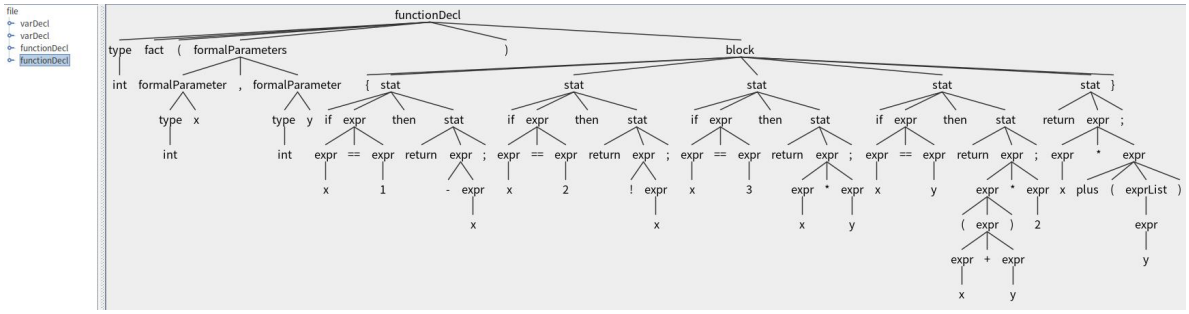
3. javac Cymbol*.java
4. grun Cymbol file -gui test1.cymbol

结果如下：









DOT 测试用例设计

DOT 的语法文件如下：

```
1.  grammar DOT;
2.  graph
3.      : STRICT? (GRAPH | DIGRAPH) id? '{' stmt_list '}' ;
4.  stmt_list
5.      : ( stmt ';' ? )* ;
6.  stmt
7.      : node_stmt
8.      | edge_stmt
9.      | attr_stmt
10.     | id '=' id
11.     | subgraph
12.     ;
13. attr_stmt
14.     : (GRAPH | NODE | EDGE) attr_list ;
15. attr_list
16.     : ('[' a_list? ']')+ ;
17. a_list
18.     : (id ('=' id)? ','?)+ ;
19. edge_stmt
20.     : (node_id | subgraph) edgeRHS attr_list? ;
21. edgeRHS
22.     : (edgeop (node_id | subgraph) )+ ;
23. edgeop
24.     : '->' | '--' ;
25. node_stmt
26.     : node_id attr_list? ;
27. node_id
28.     : id port? ;
29. port
30.     : ':' id (':' id)? ;
31. subgraph
32.     : (SUBGRAPH id?)? '{' stmt_list '}' ;
```

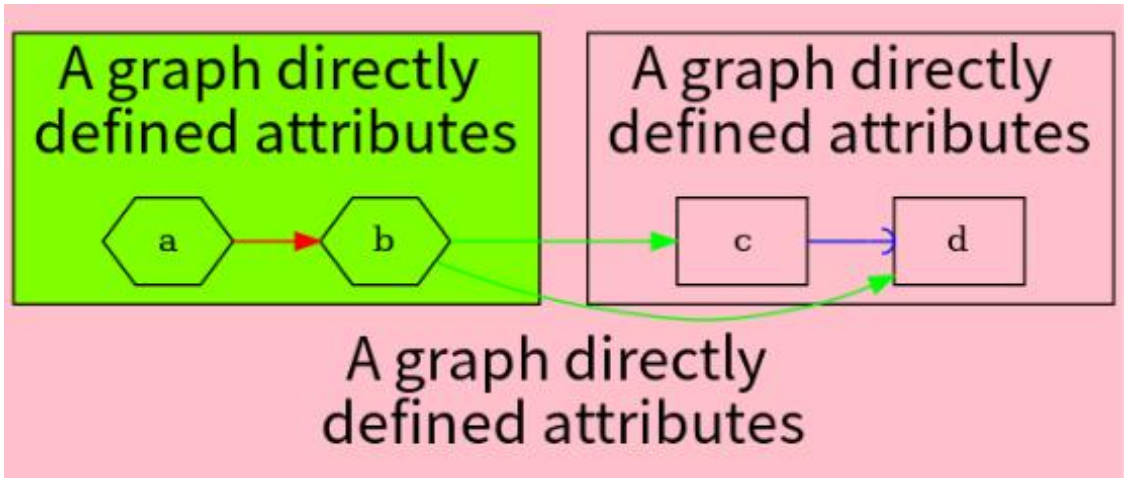
```
id
    : ID
    | STRING
    | HTML_STRING
    | NUMBER
    ;

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（test1.dot）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：

```
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
    */
}
```

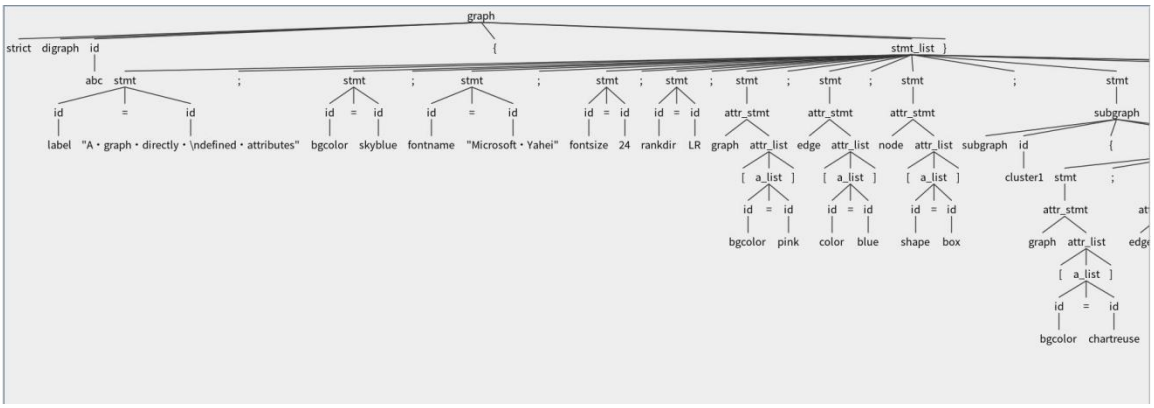
该 dot 文件生成的图结构如下：

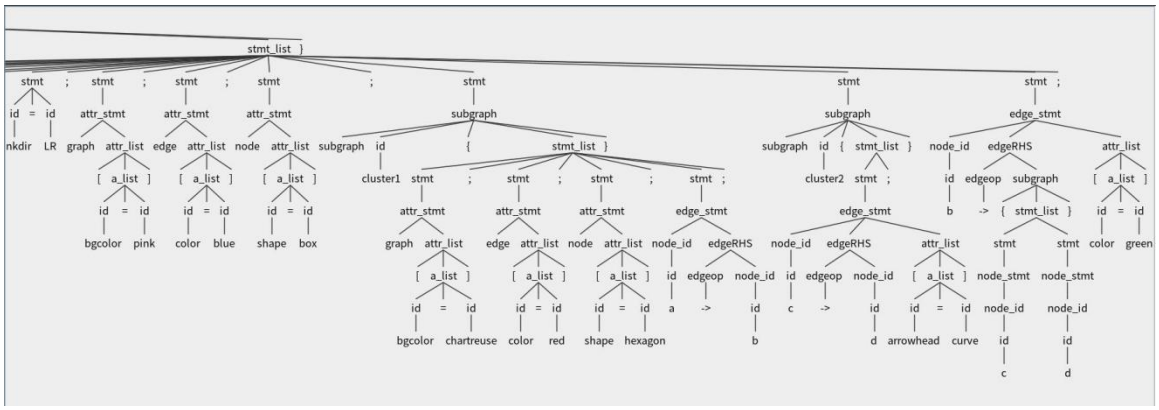


终端操作步骤：

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

结果如下：

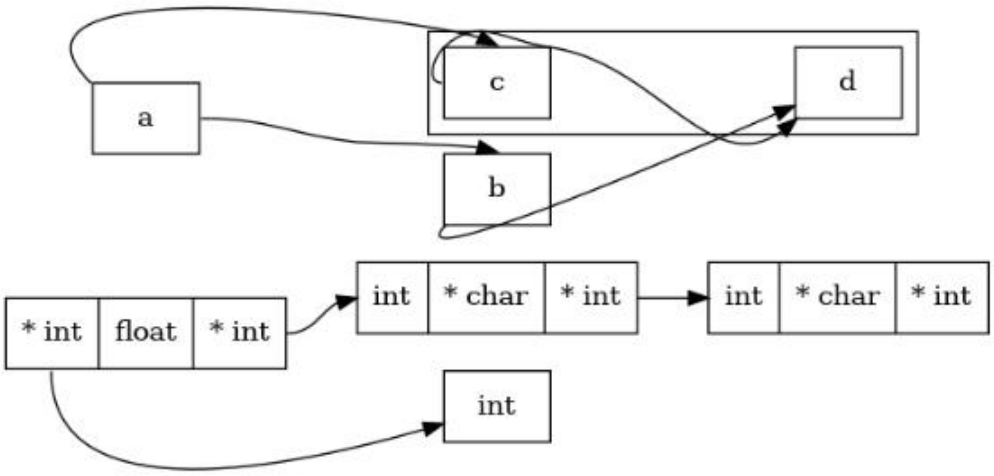




第二个测试用例（test2.dot）使用了 DOT 语言中部分的关键字和字符（包括竖杠|），并实现了罗盘端口和命名端口的语法规则形式。

```
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;
}
```

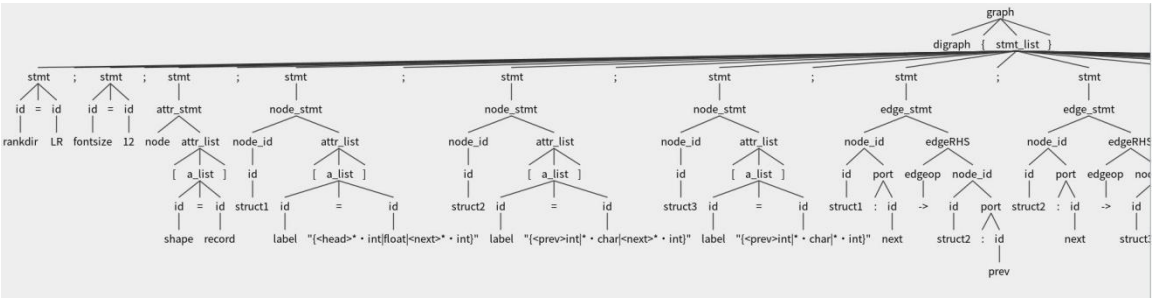
该 dot 文件生成的图结构如下：

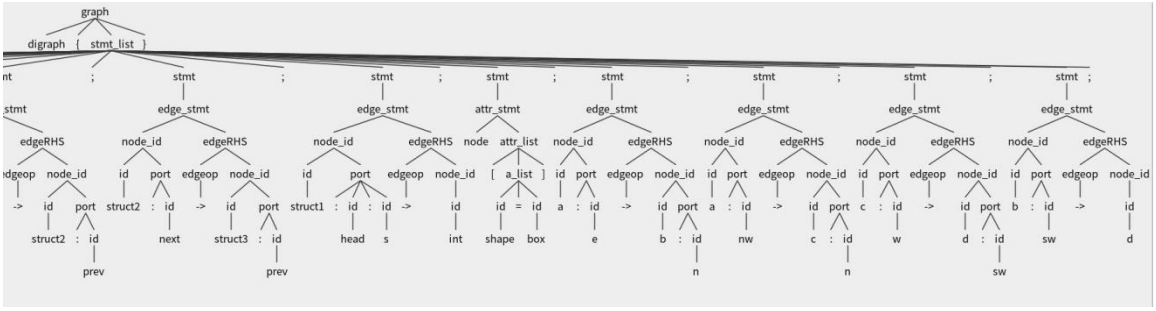


终端操作步骤：

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

结果如下：





R 测试用例设计

R 的语法文件如下：

```
1.  grammar R;
2.  prog
3.      : ( expr_or_assign (';'|NL) | NL)* EOF
4.      ;
5.  expr_or_assign
6.      : expr ('<-'|'='|'<<-') expr_or_assign
7.      | expr
8.      ;
9.  expr:   expr '["[' sublist ']' ']'  // '[[' follows Rs yacc grammar
10.     |   expr '[' sublist ']'
11.     |   expr ('::'|':::') expr
12.     |   expr ('$'|'@') expr
13.     |   <assoc=right> expr '^' expr
14.     |   ('-'|'+') expr
15.     |   expr ':' expr
16.     |   expr USER_OP expr // anything wrappedin %: '%' .* '%'
17.     |   expr ('*'|'/') expr
18.     |   expr ('+'|'-') expr
19.     |   expr ('>'|'>='|'<'|'<='|'=='|'!=') expr
20.     |   '!' expr
21.     |   expr ('&'|'&&') expr
22.     |   expr ('|'|'||') expr
23.     |   '~' expr
```

```
24.     |  expr '~' expr
25.     |  expr ('->'|'->>'|':=') expr
26.     |  'function' '(' formlist? ')' expr // define function
27.     |  expr '(' sublist ')'              // call function
28.     |  '{' exprlist '}' // compound statement
29.     |  'if' '(' expr ')' expr 'else' expr
30.     |  'if' '(' expr ')' expr
31.     |  'for' '(' ID 'in' expr ')' expr
32.     |  'while' '(' expr ')' expr
33.     |  'repeat' expr
34.     |  '?' expr // get help on expr, usually string or ID
35.     |  'next'
36.     |  'break'
37.     |  '(' expr ')'
38.     |  ID
39.     |  STRING
40.     |  HEX
41.     |  INT
42.     |  FLOAT
43.     |  COMPLEX
44.     |  'NULL'
45.     |  'NA'
46.     |  'Inf'
47.     |  'NaN'
48.     |  'TRUE'
49.     |  'FALSE'
50.     ;
51. exprlist
52.     : expr_or_assign ((';'|NL) expr_or_assign? )*
53.     |
54.     ;
55. formlist
56.     : form (',' form)*
57.     ;
58. form
59.     : ID
60.     | ID '=' expr
61.     | '...'
62.     ;
63. sublist
64.     : sub (',' sub)* ;
65. sub
66.     : expr
67.     | ID '='
68.     | ID '=' expr
69.     | 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?
```

```
116.            ;
117.        ID  :  '.' (LETTER|'_'|'.') (LETTER|DIGIT|'_'|'.')*
118.            |  LETTER (LETTER|DIGIT|'_'|'.')*
119.            ;
120.        fragment LETTER  : [a-zA-Z] ;
121.        USER_OP :  '%' .*? '%' ;
122.        COMMENT :  '#' .*? '\r'? '\n' -> type(NL) ;
123.        // Match both UNIX and Windows newlines
124.        NL      :  '\r'? '\n' ;
125.        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：

对于该测试用例的详细解读如下：

```
1.  //test1.R
2.  myString <- "Hello, World!"
3.  arbit = c(0,1,2,3)
4.  klasse <- 9+2*4/8-5 <= 8
5.  name <- !(8 >= 4 & 8 <= 9.5)
6.  name1 <- ~(-8 < 7 && +8 > 2)
7.  name2 <- 12 == 12 || 12 != 9
8.  name8 <<- 4 | 5
9.  name11 :: name12
10. name13 ::: name14
11. name15 $ name16
12. name17 @ name18
13. if (name1 == name2) name4 <- TRUE
14. round(1.5)
15. name5 <- {name1 + name2}
16. name6 <- 1
17. name7 <- 1
18. for ( name6 in 5) name7 <- name7 + 1
19. nameloop <= 1
20. while(nameloop < 10) nameloop <- nameloop + 1
21. repeat break
22. ?name2
23. print ( myString )
```

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。

终端操作步骤：

1. antlr4 R.g4
2. javac R*.java
3. 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：