如何优雅地展平你的语法树

背景

上一周进行的编译原理实验的难度并不小。虽然从实验的命名——《语法分析实验》来看我们需要进行的只是对C语言进行**语法**分析，但我们实际需要做的工作远大于此。

如果使用`antlr`进行解析而不是使用递归下降器法，那么编译的关键步骤在本次实验涉及了三项：词法分析、语法分析、以及本文主要阐述的**抽象语法树的构造**（语法制导翻译的其中一个步骤）。一个实验能从**词法**分析开始直到迈入**语义**分析，跨度可谓十分巨大。

事实上真正的"语法分析"工作事实上已经由`antlr`生成的语法树生成完毕，令包括我在内的许多同学感到比较困难的仅是最后一步`json`的构造。

根据我的观察，无论是使用递归下降器、手搓字符串还是AST节点法；无论语法的BNF表示的简单与复杂；无论语法定义是否接近老师的目标节点；任何人写出的最终代码行数都不少且复杂度较高。这是主要因为目标对应的`json`树与自己设定的`antlr`树是完全不同的结构。

通过我的思考，我认为本次实验的难度可规约为一个原因：本次实验的实质是在语法分析的基础上更进一步——将CST树转化为AST树。

语法树 or 抽象语法树?

让我们观察下面句子：

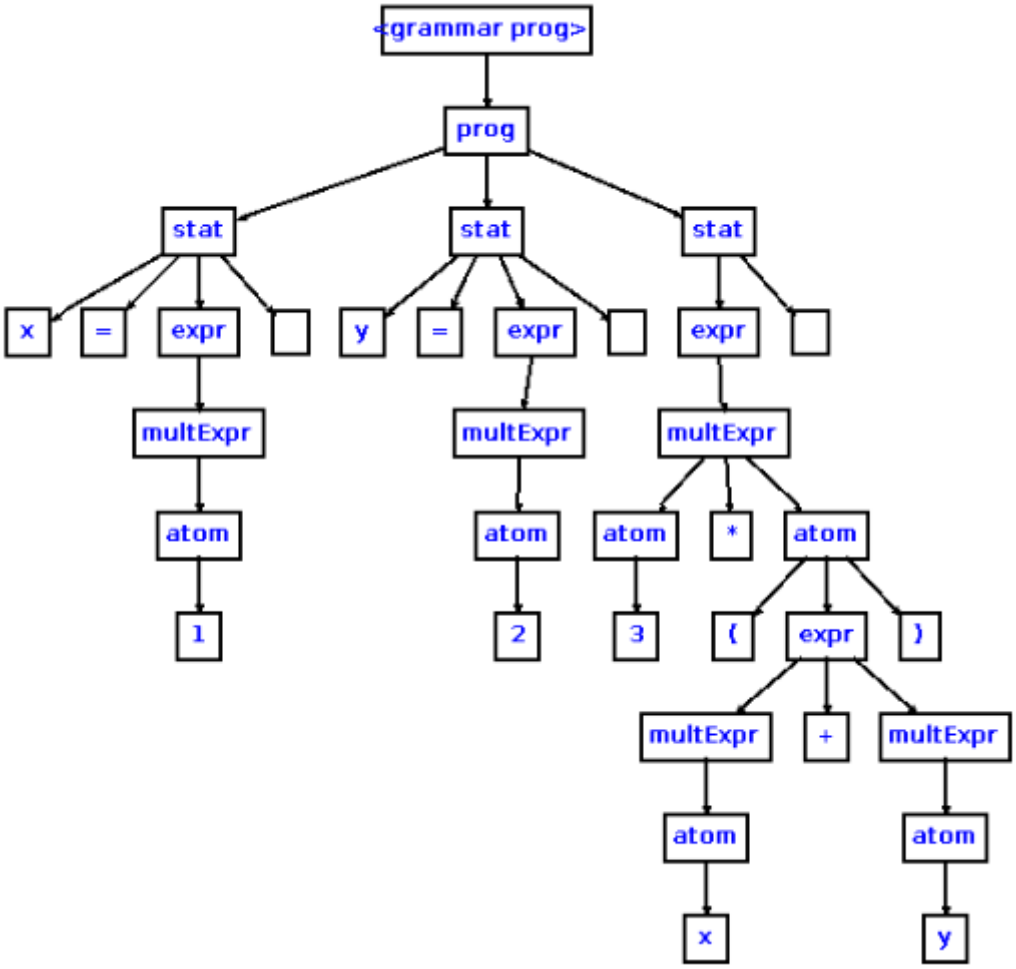
```
x=1
y=2
3*(x+y)
```

如果我们用文法

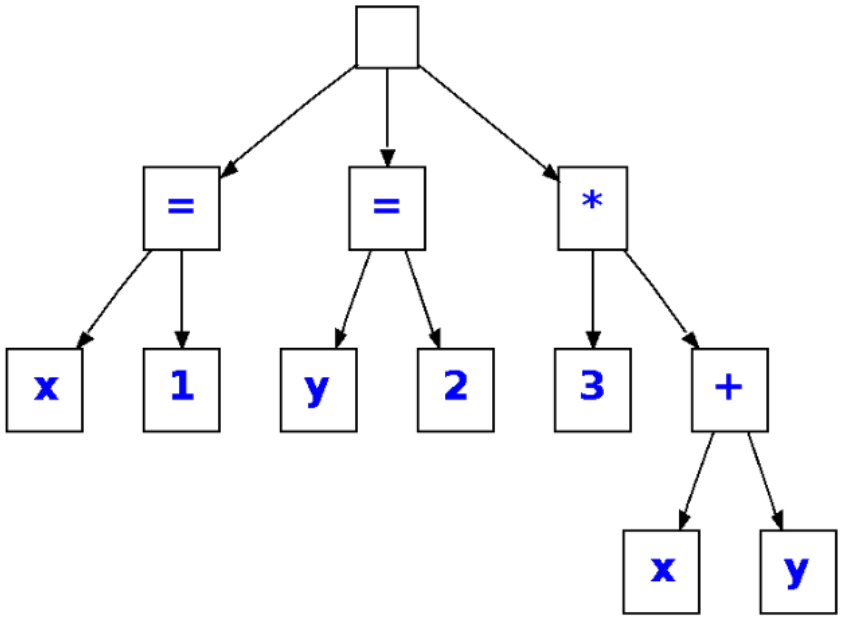
```
prog    :   ( stat )+ ;

stat    :   expr NEWLINE        -> expr
        |   ID '=' expr NEWLINE -> ^('=' ID expr)
        |   NEWLINE             ->
        ;

expr    :   multExpr (( '+'^ | '-'^ ) multExpr)*
        ;

multExpr
        :   atom ('*'^ atom)*
        ;

atom    :   INT
        |   ID
        |   '('! expr ')'!
        ;

ID      : ('a'..'z' | 'A'..'Z' )+ ;
INT     : '0'..'9'+ ;
NEWLINE : '\r'? '\n' ;
```

```
WS      : ( ' ' | '\t' )+ { skip(); } ;
```

进行解析，则它可生成如下的语法树：



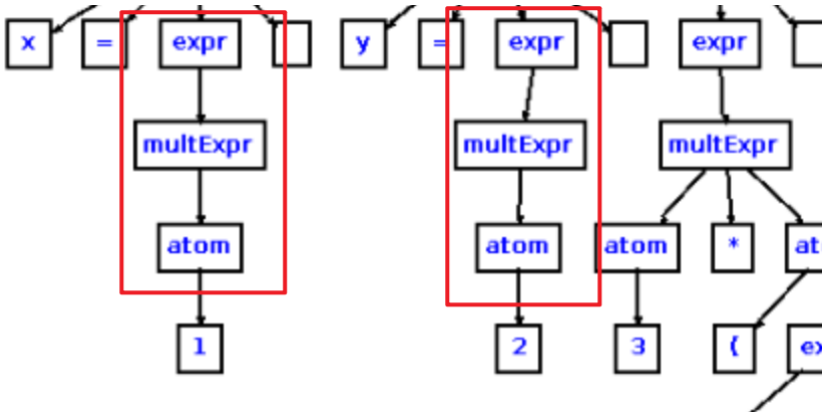
看起来很正确，但其实这株树还有一个更简洁的表达方法：



你会发现，下面的这株树比上面的语法树要平坦、简洁许多，于此同时也保证了语义（运算优先级）的不变。

前面的这颗树即通过我们文法规则生成的**语法树**(Parser Tree)，也可以被叫做**CST**(Concrete Syntax Tree)——因为它非常“忠实”(concrete)地还原了我们的构造的文法规则。

后面的这颗树即**抽象语法树**(**A**bstract **S**yntax **T**ree)，即**AST**，它减少了许多本来不必要的信息。



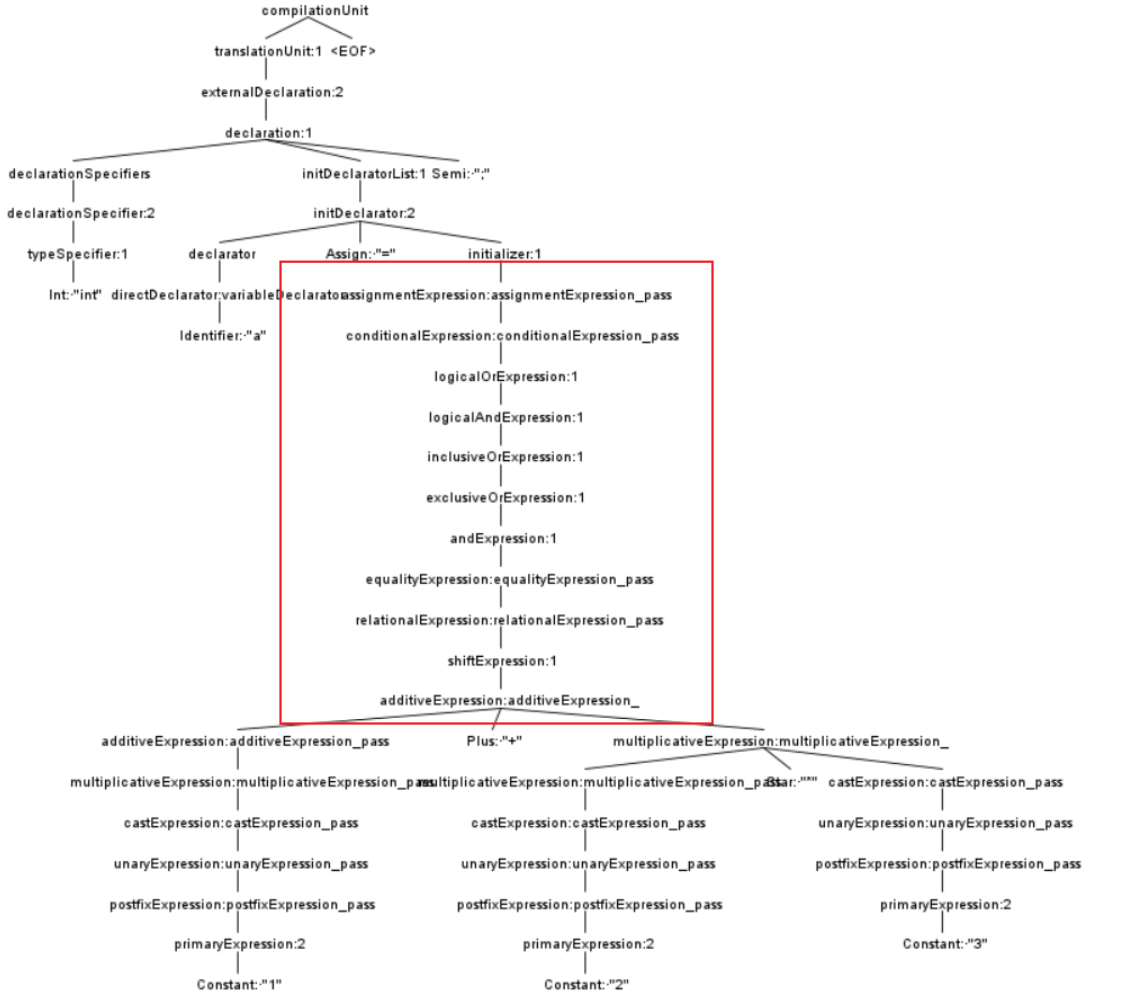
上图用红框框出的，就是“不必要的信息”。

CST完全对应于文法规则，每一节点的分叉少但树的深度大，冗余信息多。AST是CST的简化，每一个节点分叉多，树更为扁平，使得后续过程能方便的获取更多信息。（感觉有点类似B树对于外存的作用）

考察C语句：

```
int a=1+2*3;
```

使用我定义的`antlr`规则（基于C11）进行解析，即使是这么一个简单的表达式，都会生成如下图所示的极其吓人的递归深度：



产生这个的问题主要是它**过于忠诚**。

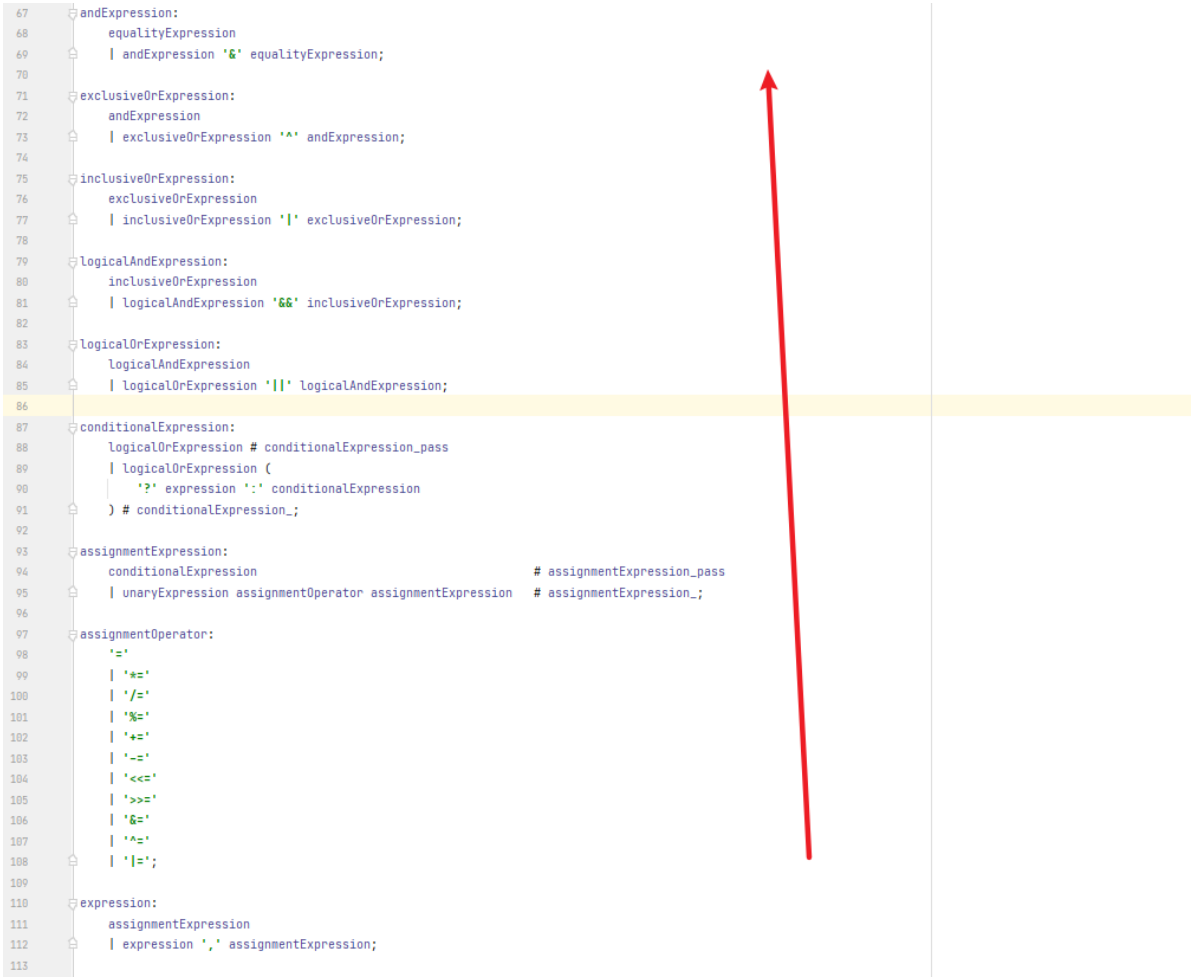
无论最终的`Expression`有多么简单，它总是要经过：

`expression` -> `assignmentExpression` -> `conditionalExpression` -> `logicalOrExpression` -> `logicalAndExpression` -> `inclusiveOrExpression` -> `exclusiveOrExpression` -> `andExpression` -> `equalityExpression` -> `relationalExpression` -> `shiftExpression` -> `additiveExpression` -> `multiplicativeExpression` -> `castExpression` -> `unaryExpression` -> `postfixExpression` -> `primaryExpression`

十分冗长的链条。通过层层嵌套，一个常量或者变量`identifier`才能找到属于它的叶子结点。

但如果我们考察目标生成对象——**抽象语法树**树（通过`json`格式展现），我们发现它总能以较浅的方式生成对应的语法树表达式：

令表达式递归深度较高的罪魁祸首是文法规则的层层嵌套：



```
andExpression:
    equalityExpression
    | andExpression '&' equalityExpression;

exclusiveOrExpression:
    andExpression
    | exclusiveOrExpression '^' andExpression;

inclusiveOrExpression:
    exclusiveOrExpression
    | inclusiveOrExpression '|' exclusiveOrExpression;

logicalAndExpression:
    inclusiveOrExpression
    | logicalAndExpression '&&' inclusiveOrExpression;

logicalOrExpression:
    logicalAndExpression
    | logicalOrExpression '||' logicalAndExpression;

conditionalExpression:
    logicalOrExpression # conditionalExpression_pass
    | logicalOrExpression (
        '?' expression ':' conditionalExpression
    ) # conditionalExpression_;

assignmentExpression:
    conditionalExpression                                   # assignmentExpression_pass
    | unaryExpression assignmentOperator assignmentExpression   # assignmentExpression_;

assignmentOperator:
    '='
    | '*='
    | '/='
    | '%='
    | '+='
    | '-='
    | '<<='
    | '>>='
    | '&='
    | '^='
    | '|=';

expression:
    assignmentExpression
    | expression ',' assignmentExpression;
```

语法树的递归

除了**嵌套**外，造成冗余信息产生的因素还有**递归**。

在解析一个文法的列表语句（逗号表达式）时递归十分常见：如`int a,b,c,d`或`pinrf("%d%d",a,c)`等句子。

考察文法：

```
initDeclaratorList: initDeclarator | initDeclaratorList ',' initDeclarator;
```

我们可以发现表达式是左递归的。`initDeclaratorList`通过采用递归自身扩充列表，最后再以`initDeclarator`存放终结符，形成一个列表表达式。

```
initDeclaratorList: initDeclarator  | initDeclaratorList ',' initDeclarator;
```

如果使用这个规则解析`int a,b,c,d,e,f,g`则会生成一个左递归十分深的语法树：

但如果不用递归，我们直接在顶层节点`declaration`中添加一个列表，将递归项目`initDeclarationList`的所有叶子节点直接挂载到`declaration`中的List而不是上一层节点的`initDeclarationList`中，就可以将这个左递归树展平而生成一颗**抽象语法树**。对应**抽象语法树**的`json`表示如下（只有一层列表）：

```
"initLists": [
                {
                    "type": "InitList",
                    "declarator": {
                        "type": "VariableDeclarator",
                        "identifier": {
                            "type": "Identifier",
                            "value": "a",
                            "tokenId": 1
                        }
                    },
                    "exprs": []
                },
                {
                    "type": "InitList",
                    "declarator": {
                        "type": "VariableDeclarator",
                        "identifier": {
                            "type": "Identifier",
                            "value": "b",
                            "tokenId": 3
                        }
                    },
                    "exprs": []
                },
              // .....
                {
                    "type": "InitList",
                    "declarator": {
                        "type": "VariableDeclarator",
                        "identifier": {
                            "type": "Identifier",
                            "value": "g",
                            "tokenId": 13
```

```
                    }
                },
                "exprs": []
            }
        ]
```

而如果使用还原语法规则的**语法树**CST，它的`json`可能就是这样的（注意列表的递归）：

```
"initLists": [
            {
                "type": "InitList",
                "declarator": {
                    "type": "VariableDeclarator",
                    "identifier": {
                        "type": "Identifier",
                        "value": "a",
                        "tokenId": 1
                    }
                    },
                    "exprs": {
                        "type": "InitList",
                        "declarator": {
                            "type": "VariableDeclarator",
                            "identifier": {
                                "type": "Identifier",
                                "value": "b",
                                "tokenId": 1
                                },
                            //....
                                    "exprs": {
                                       "type": "InitList",
                                       "declarator": {
                                           "type": "VariableDeclarator",
                                           "identifier": {
                                               "type": "Identifier",
                                               "value": "c",
                                               "tokenId": 1
                                           },
                },
            ]
```

AST有着相对简单的结构，语法树较浅，使得在后续语义分析时可以很方便地用类似数组遍历的方式就能取到需要的节点的值；但后者CST取值时需要再次进行树的中序遍历

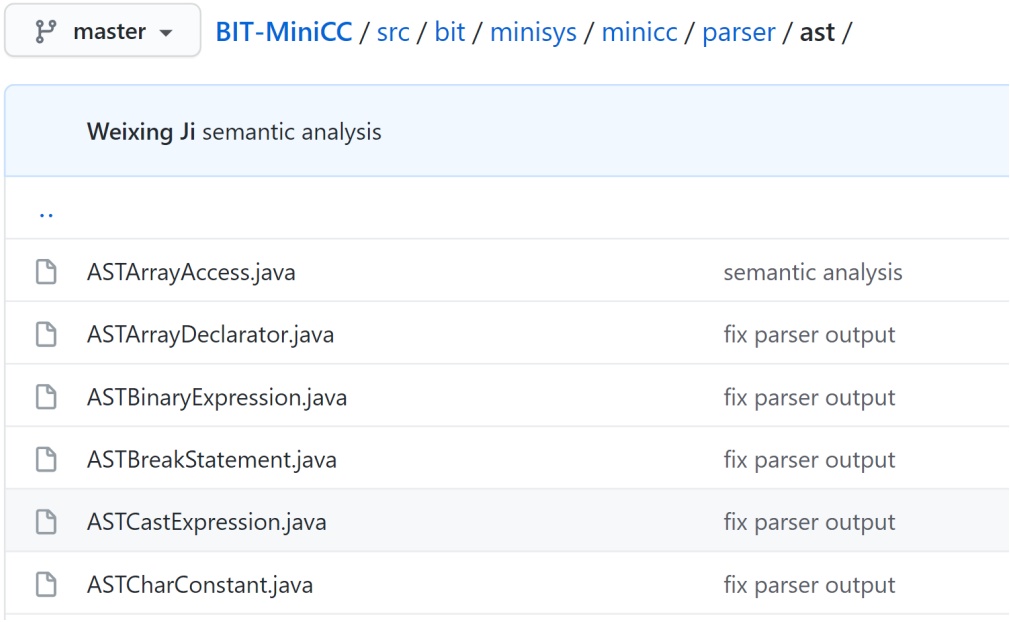
类似BNF的表述文法使用嵌套和递归很大程度上的增强了文法规则表达的简洁性，但是也使得在解析文法的过程中会引入许多无意义的节点（嵌套、递归），对应的CST在后续过程中也很难使用。因此，我们的任务就是消除这些冗余节点，”**展平**“我们的语法树，使其成为一棵抽象语法树。

如何优雅地从CST生成AST?

下面通过结合实验实例给出算法的描述

实验流程/工具/过程/预置条件概述

目标抽象语法树的节点已经预定义好，即为[BIT-MiniCC/src/bit/minisys/minicc/parser/ast at master · jiweixing/BIT-MiniCC (github.com)](https://github.com/jiweixing/BIT-MiniCC/tree/master/src/bit/minisys/minicc/parser/ast)中的AST类：



AST类经过实例化后得到对应的节点，**我们将AST节点互相连接，生成一个"JAVA对象构成的AST树"**，最后将这棵树的根节点送入`jackson`中自动生成`json`字符串并进行输出，所得结果即为所需\。

为了将AST节点相互连接，我们需要以某种方式遍历`antlr`的语法树，并在遍历语法树的过程中按某种规则挂载AST节点，构造对应的抽象语法树——那么，如何使这个过程变得优雅呢?

算法概述

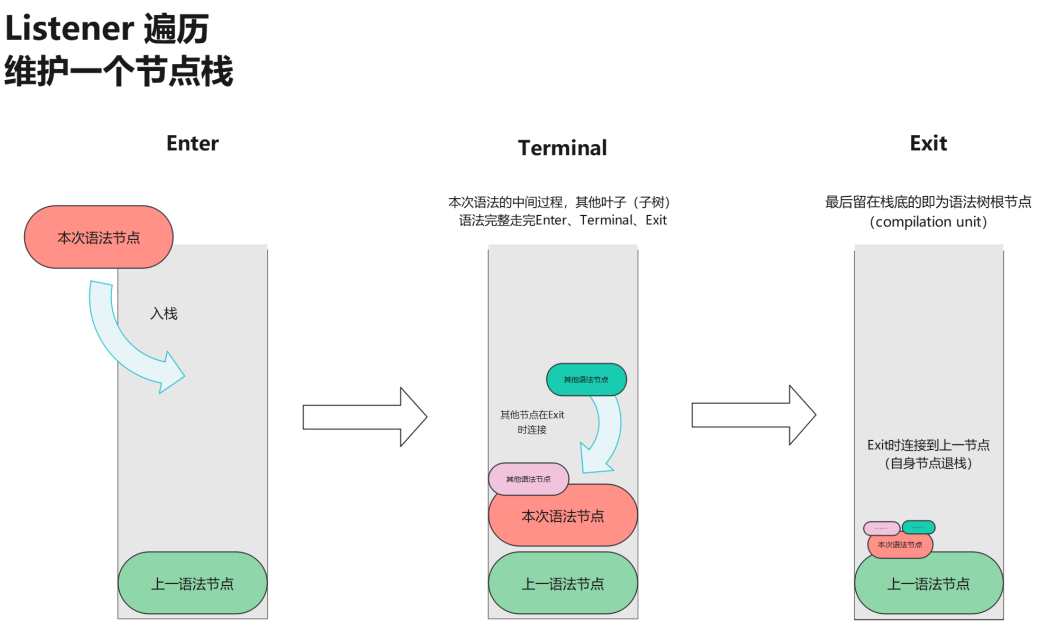
遍历`antlr`节点主要有`visitor`和`listener`的方法。根据我观察，周围的大部分人都选择了使用`visitor`，但我认为使用`visitor`十分不方便：

- 流程复杂。`visitor`进入每一个节点时，都要自己定义遍历其他节点的法则。
  - 作为对比，我的方法是存粹的自动遍历。
- 代码耦合度高。遍历规则和已有语法树高度相关，倘若在后续过程中发现语法树有问题，改动语法树的同时极有可能要大幅度改动已有代码。
  - 作为对比，理论上我的方法在新增一个节点时，可能只需要手工添加3行代码。（详见后续）
- 拓展性差。如果使用`visitor`方法，许多同学会出现这种情况——自己定义的词法规则越完善，反而工作量越大，实现的C语言特性越多，偏离目标的文法规则越远，`visitor`的遍历法则就越困难，随着节点的加入代码量甚至可能呈几何形增长。
  - 作为对比，理论上我的方法定义的节点和文法规则越全，反而可能会更简单，或起码复杂度不会产生变化。
- 代码量大，维护难度高，debug困难。
  - 作为对比，我的方法在最坏情况下只是漏挂节点，并且代码处于一直可运行的状态，后续语法节点的实现不会令原有代码产生漏洞。

`visitor`法在我看来，相当于自己再手动定义令一株树。事实上我们的`antlr`树本来就已经定义好了，那么就不应该自己再定义另一颗`visitor`，手动再新增许多遍历法则，这样是十分没必要的。我们应该直接使用自动遍历的方法，在自动遍历的过程中进行AST节点的挂载。因此相比于手动遍历的`visitor`，我选用了自动遍历的`listener`。

这个方法我称为树的“模拟”。通过引入一个栈保存现场，使得在遍历一株树的时候可以正确的挂载节点模拟出另一株完全异构的树——只要遍历的树的文法规则是目标树的“超集”，理论上可以用它模拟出任何异构的树。

算法介绍



© 王梓丞2021

算法步骤如下：

1. 进入语法树的节点时，新建对应的AST节点，将AST节点推入栈中
2. 离开语法树本节点时，对应AST节点出栈（并维护一个对它的引用）。将本AST节点挂载到栈顶的节点（即父语法节点）对应位置。

每一个节点进入时只用关注父节点的情况，由于栈保存了现场信息，因此节点退栈时栈顶总是语法树的父节点。在节点挂载时，需要通过条件判断判断父亲节点的类型，通过此类型可以获得信息，以得知本节点该挂载在哪里。

算法很简单，但或许有点抽象，接下来结合实例来说明：

以程序

```
int main()
{
    int a;
}
```

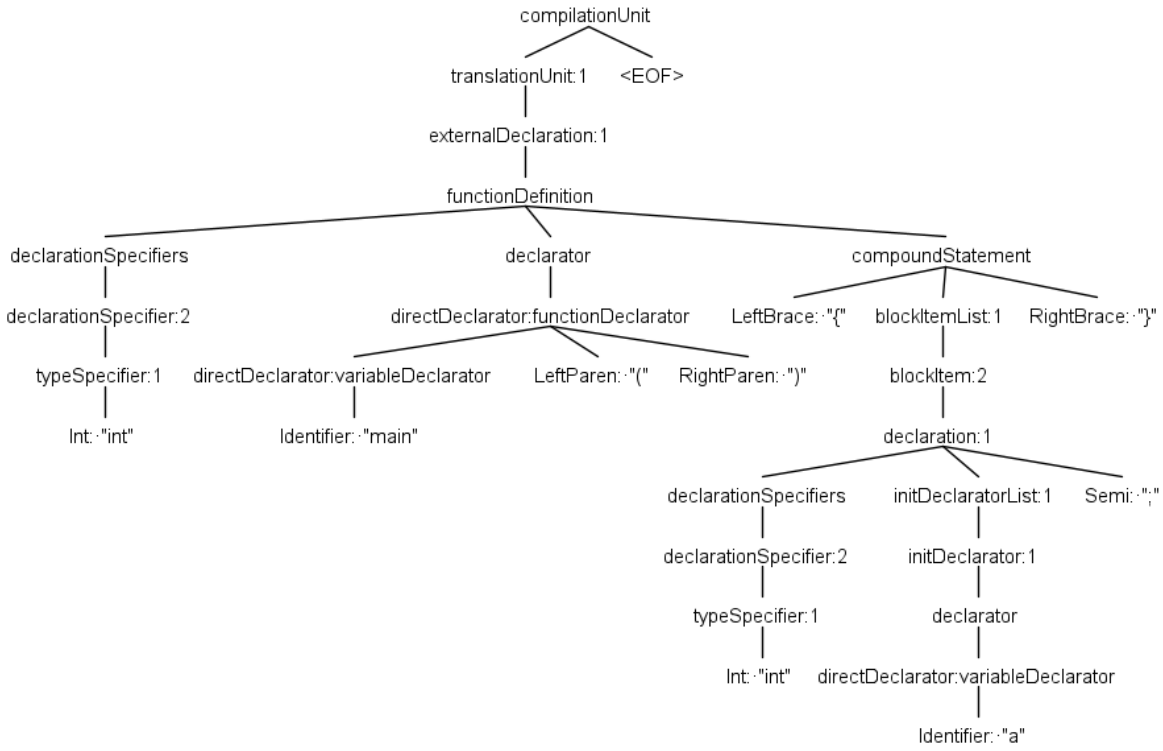
为例。

其生成的`json`如下（AST）：

```
{
    "type": "Program",
    "items": [
        {
```

```
            "type": "FunctionDefine",
            "specifiers": [
                {
                    "type": "Token",
                    "value": "int",
                    "tokenId": 0
                }
            ],
            "declarator": {
                "type": "FunctionDeclarator",
                "declarator": {
                    "type": "VariableDeclarator",
                    "identifier": {
                        "type": "Identifier",
                        "value": "main",
                        "tokenId": 1
                    }
                },
                "params": []
            },
            "body": {
                "type": "CompoundStatement",
                "blockItems": [
                    {
                        "type": "Declaration",
                        "specifiers": [
                            {
                                "type": "Token",
                                "value": "int",
                                "tokenId": 5
                            }
                        ],
                        "initLists": [
                            {
                                "type": "InitList",
                                "declarator": {
                                    "type": "VariableDeclarator",
                                    "identifier": {
                                        "type": "Identifier",
                                        "value": "a",
                                        "tokenId": 6
                                    }
                                },
                                "exprs": []
                            }
                        ]
                    }
                ]
            }
        }
    ]
}
```


对应的语法树如下：



FunctionDefine	// int main(){ }

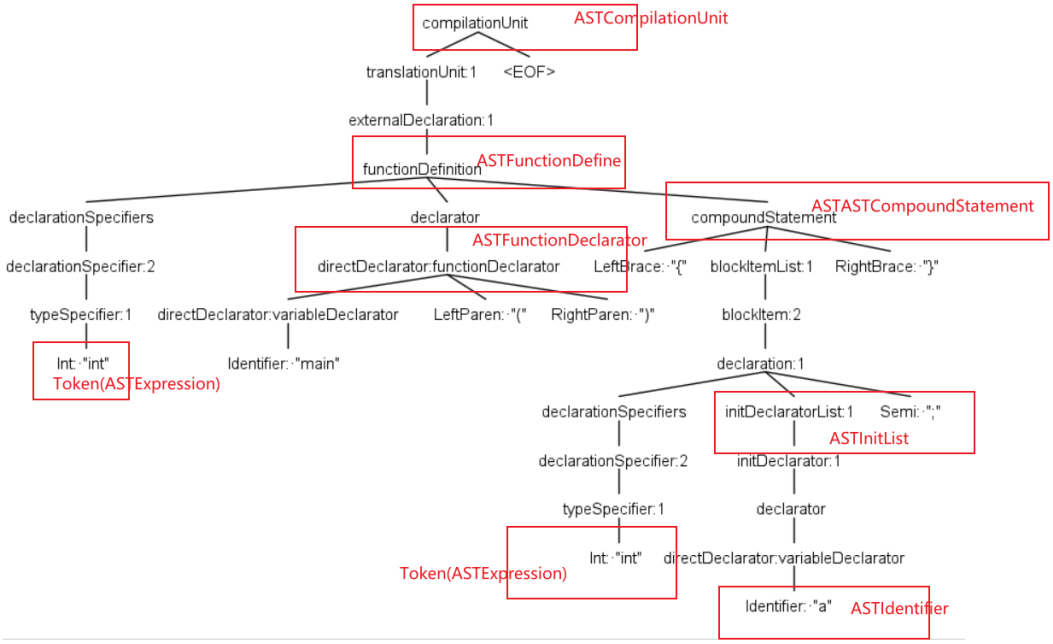
```
{
    "type": "FunctionDefine",
    "specifiers": [
        {
            "type": "Token",
            "value": "int",
            "tokenId": 0
        }
    ],
    "declarator": {
        "type": "FunctionDeclarator",
        "declarator": {
            "type": "VariableDeclarator",
```

通过观察语法分析实验的文档我们可知，其对应经过的AST节点，以及他们的层级关系是：

`FunctionDefine`

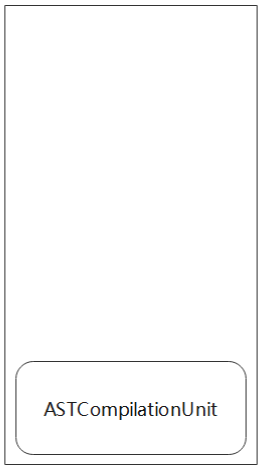
- `Token`
- `FunctionDeclarator`
- `CompoundStatement`
  - `Declaration`
    - `VariableDeclarator`

针对这些AST节点，我们目前需要做的就是找到它们与我们语法树节点的对应，并修改它们的`listener`方法，使其可以正确挂载。

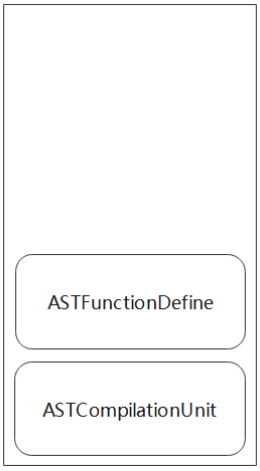


图示即为语法树节点对应的AST节点。在这个过程中本算法的简要流程如下：

1. 进入语法树的`CompilationUnit`节点，执行对应`listener`的`enter`方法，将对应AST节点压栈



2. 进入`functionDefine`节点，执行对应`listener`的`enter`方法，将对应AST节点压栈



3. 接着根据语法树的中序遍历规则进入了左部的第一个`Token`节点(其值是`int`)。这里有一点特殊，就是任何终结符的节点（`antlr`的以大写开头的词法规则）都是没有`enter`和`exit`方法的，它们使用`visitTerminal`。但总而言之结果是大同小异的，在进入终端节点、`listener`调用`visitTerminal`时，相当于“压栈、退栈、挂载节点”合并为了一个操作——这个时候将`Token`对应的AST节点`ASTExpression`实例化后，判断栈顶元素（父亲节点）后直接挂载。

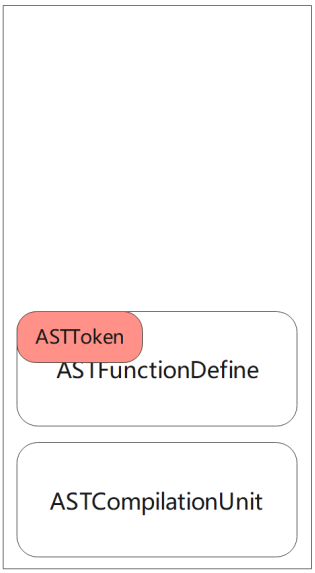
FunctionDefine	// int main() { }

```
{
    "type": "FunctionDefine",
    "specifiers": [
        {
            "type": "Token",
            "value": "int",
            "tokenId": 0
        }
    ],
    "declarator": {
        "type": "FunctionDeclarator",
        "declarator": {
            "type": "VariableDeclarator",
```

比如此时检测到栈顶元素为`ASTFunctionDefine`，于此同时，它的`specifiers`属性刚好对应于终端属性的`ASTToken`，那么终端节点就可以认为这个属性是需要自己挂载的挂载点。执行代码：

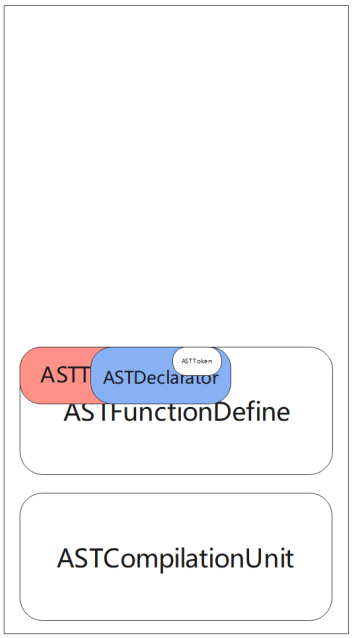
```
if (parentNode.getClass() == ASTFunctionDefine.class) //判断父亲节点类型
{
    ((ASTFunctionDefine) parentNode).specifiers//父亲节点的挂载点
                        .add(new ASTToken(node.getSymbol().getText(),
node.getSymbol().getTokenIndex()));
                        //本节点实例化并挂载
}
```

就可以实现本次节点的挂载：

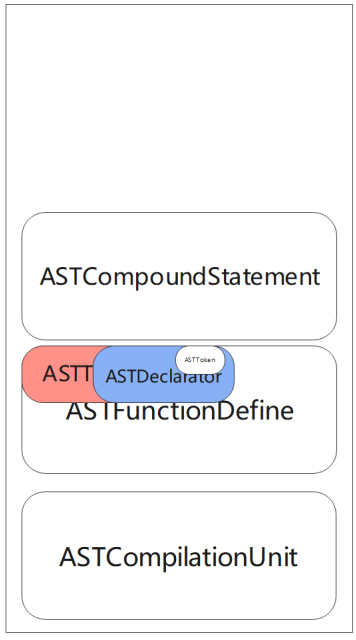


注意，“节点的挂载”属于一种“内聚”的关系，当子节点挂载后，子节点本身并不在栈内，而是“内聚”到了父节点的属性上，此时栈顶的元素必定是父节点（内聚了一个或多个子节点）而不再是子节点本身。

4. 根据语法树的中序依次执行`ASTFunctionDeclarator`的入栈、`ASTFunctionDeclarator`中终端节点`Identifier`的挂载、`ASTFunctionDeclarator`挂载到`ASTFunctionDefine`的`declarator`属性就可以得到以下的栈：

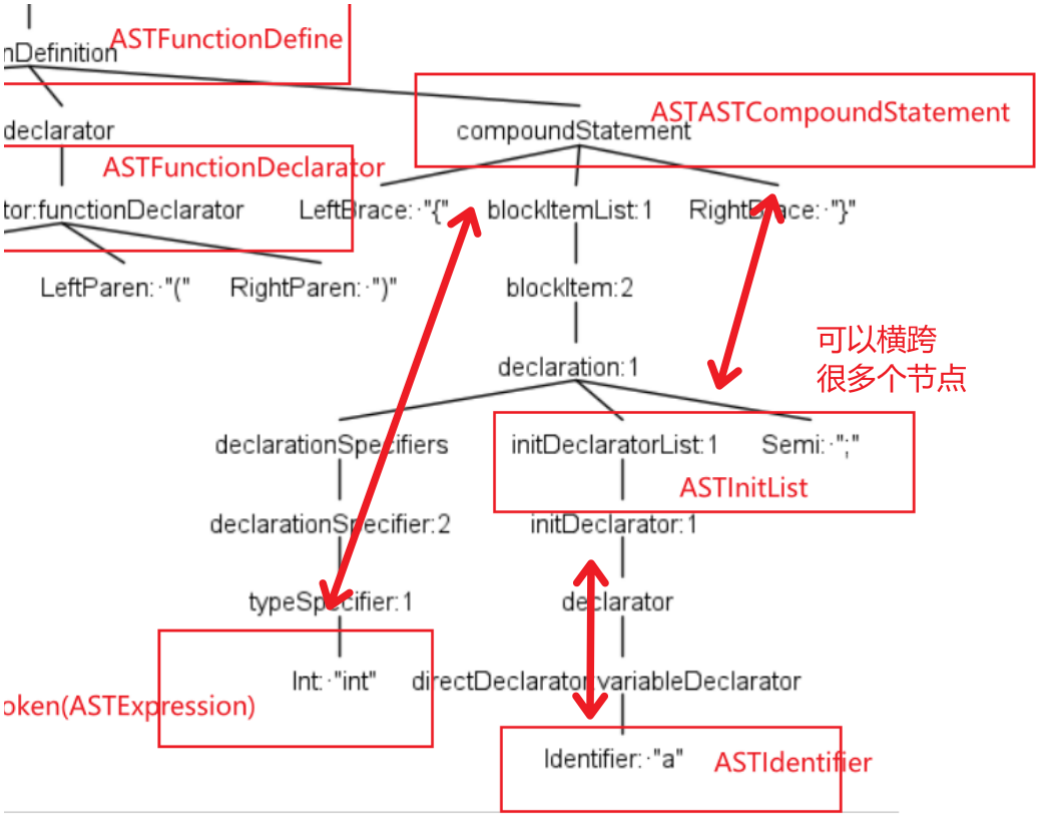


5. 接着`ASTCompoundStatement`入栈，正如上面描述的，其兄弟节点和兄弟节点的子节点已经退栈并挂载到对应的父节点上，因此栈顶顶层的元素必定是其父节点`ASTFunctionDefine`。



6. 最后再执行`ASTCompundStatement`的子节点挂载、退栈等操作，最后等`ASTFunctionDefine`也挂载后，栈底剩下的最后一个元素`ASTCompilationUnit`即为我们抽象语法树的根。将其送入`json`自动化生成工具即能得到正确答案。

注意，AST节点和语法树节点并不完全一一对应，并且它们相互之间可以不必挨着——比如任何一个`Token`都可以离其语法树上的父节点很远，可以横跨好几个冗余节点。



这也是本算法的优势，比如如果日后对文法有需求，需要在`type_Sepcifier`和`declarationSpecifiers`间再嵌套一些节点（比如一些编译的标识符），也并不会影响`int`作为`Token`对其AST树的父节点`ASTFunctionDefination`的挂载——因为在这个过程中冗余节点会被自动“规约掉”。这个特性在后文提到的`Expression`表达式的多层嵌套中尤为重要。

接下来从原理方面讲述算法的可靠性

递归消除原理

造成递归的主要原因是语法中的列表。但其实任何有列表性质的语法节点，都一定有个顶层节点。

比如对于`blockItem`：

```
blockItemList: blockItem | blockItemList blockItem;
```

大括号中的语句列表，其顶层一定是`compoundStatement`：

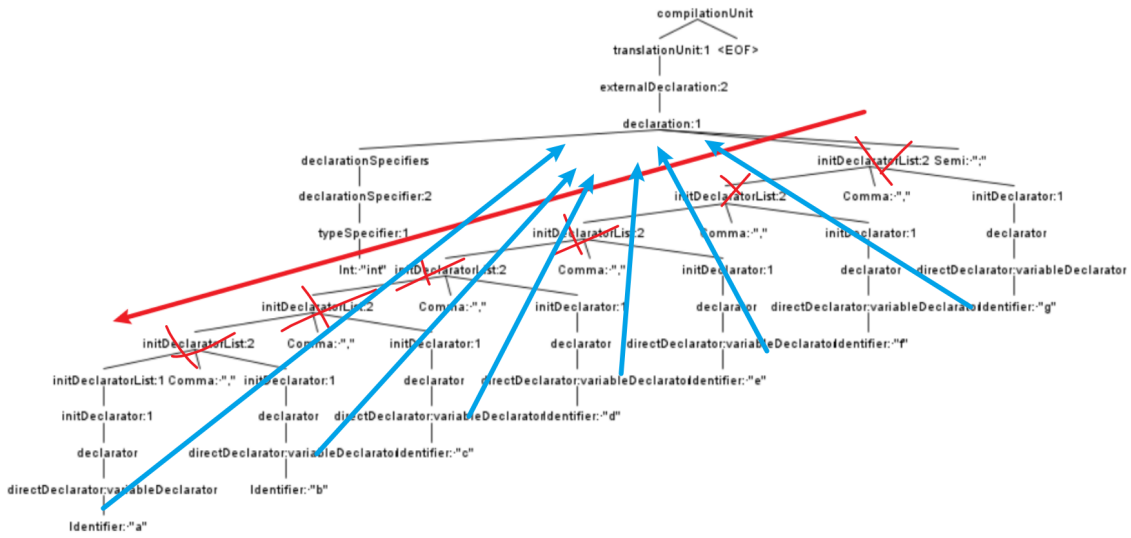
```
compoundStatement: '{' blockItemList? '}';
```

那么我们可以选取一种策略：**忽略所有的*List节点**，只关注它们的子节点。让子节点判断栈顶的顶层元素，直接将自身挂载到顶层节点的列表属性中。

一个简略版本的代码可以写在下面：

```
@Override
public void exitStatement(StatementContext ctx) {
    thisNode = nodeStack.pop();
    parentNode = nodeStack.peek();
    if (parentNode.getClass() == ASTCompoundStatement.class) {
         if (((ASTCompoundStatement) parentNode).blockItems == null) {//注意判断父亲节点的列表是否有实例化
                ((ASTCompoundStatement) parentNode).blockItems = new LinkedList<ASTNode>();
            }
        ((ASTCompoundStatement) parentNode).blockItems.add(thisNode);//直接把节点本身，而不是节点列表挂到顶层节点
    }
}
```

上述代码引入`if (((ASTCompoundStatement) parentNode).blockItems == null)`的原因是，本次作业中节点的属性列表默认是不初始化的.....

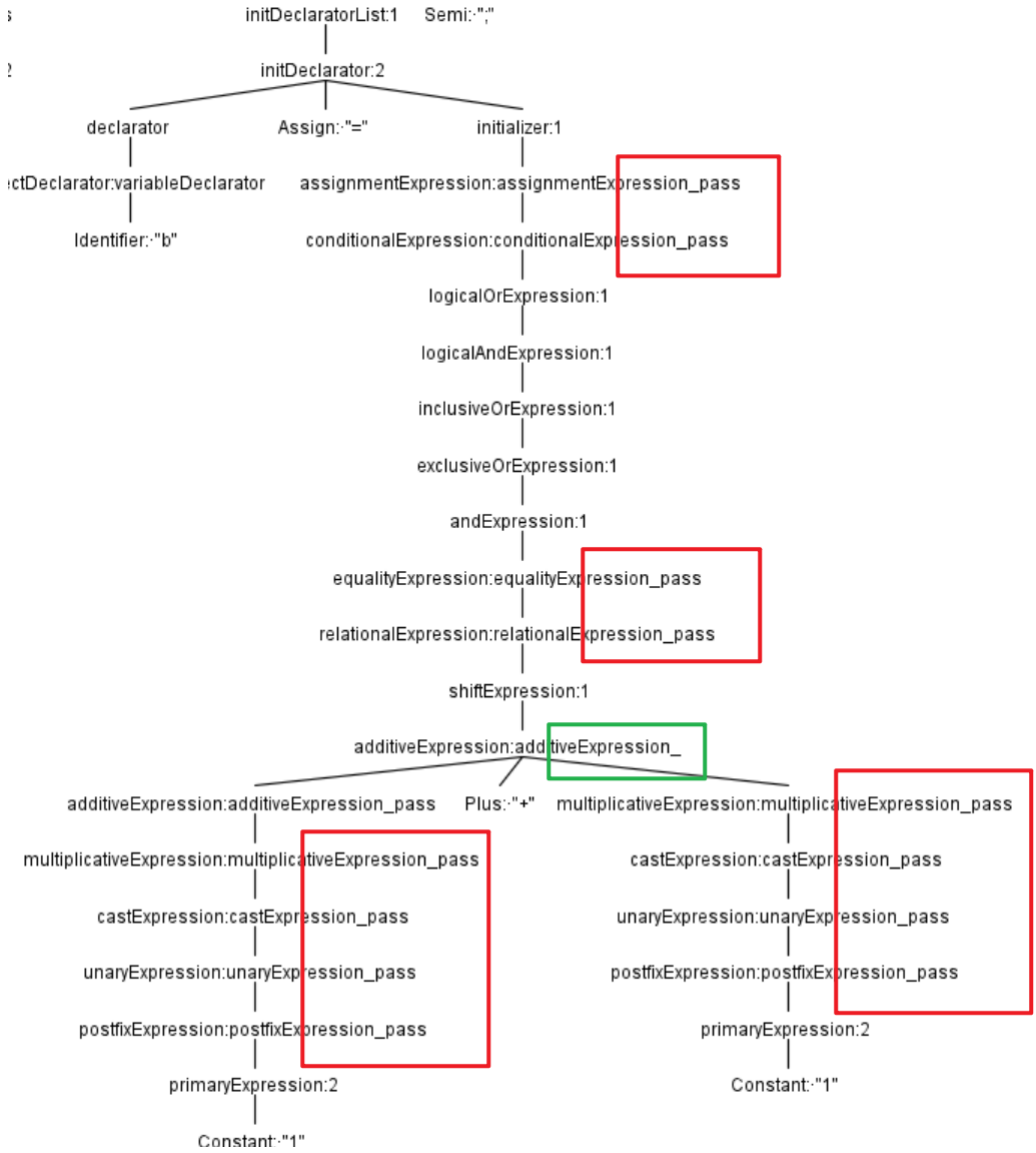


对于前文的初始化列表的嵌套，在本算法下，相当于每个`initDeclatationList`被自动pass掉，到终端节点时直接检测栈顶节点，将自身挂载，直接消除了需要编程者自己思考树的递归问题。

嵌套消除原理

深层嵌套的典型是表达式

比如即使是`int a=1+1;`这个句子，都会由于文法的优先级定义，产生很长的递归。



但如果仔细观察发现，在我定义的语法树上，许多节点定义上了`_pass`标签

```
multiplicativeExpression:
    castExpression                                  #
multiplicativeExpression_pass
    | multiplicativeExpression '*' castExpression   # multiplicativeExpression_
    | multiplicativeExpression '/' castExpression   # multiplicativeExpression_
    | multiplicativeExpression '%' castExpression   # multiplicativeExpression_;

additiveExpression:
    multiplicativeExpression                            #
additiveExpression_pass
    | additiveExpression '+' multiplicativeExpression   # additiveExpression_
    | additiveExpression '-' multiplicativeExpression   # additiveExpression_;

shiftExpression:
    additiveExpression
    | shiftExpression '<<' additiveExpression
    | shiftExpression '>>' additiveExpression;

relationalExpression:
    shiftExpression                                 # relationalExpression_pass
    | relationalExpression '<' shiftExpression  # relationalExpression_
    | relationalExpression '>' shiftExpression  # relationalExpression_
    | relationalExpression '<=' shiftExpression # relationalExpression_
```

```
    | relationalExpression '>=' shiftExpression # relationalExpression_;

equalityExpression:
    relationalExpression                              # equalityExpression_pass
    | equalityExpression '==' relationalExpression  # equalityExpression_
    | equalityExpression '!=' relationalExpression  # equalityExpression_;
```

这利用到了`antlr`的表达式的特性——别名。通过别名`#`我们可以控制`listener`的入口，最后本节点的`listener`会以别名，而不是节点的`节点名:`命名

对于一系列的嵌套节点，常常有**无意义**和**有意义**之别。

```
postfixExpression:
    primaryExpression                                       # postfixExpression_pass
    | postfixExpression '[' expression ']'                  # arrayAceess_
    | postfixExpression '(' argumentExpressionList? ')' # functionCall_
    | postfixExpression '.' Identifier                      # postfixExpression_
    | postfixExpression '->' Identifier                     # postfixExpression_
    | postfixExpression '++'                                # postfixExpression_
    | postfixExpression '--'                                # postfixExpression_;

unaryExpression:
    postfixExpression               # unaryExpression_pass
    | '++' unaryExpression          # unaryExpression_
    | '--' unaryExpression          # unaryExpression_
    | unaryOperator castExpression  # unaryExpression_
    | 'sizeof' unaryExpression      # unaryTypename_
    | 'sizeof' '(' typeName ')'     # unaryTypename_;
```

考察上述文法：`unaryExpression`中的`unaryExpression -> postfixExpression`嵌套关系，就属于“无意义”节点。因为`unaryExpression`在这中间并没有做任何事情，它只是将自己的“职责”推给了子节点`postfixExpression`。这样，我们就认为此处的`unaryExpression`没有携带任何有意义的信息，是属于冗余节点，我们将其命名为`unaryExpression_pass`即可，并且在`Listener`中对`enterUnaryExpression_pass`和`exitUnaryExpression_pass`防止不管即可。

对于`unaryExpression -> '++' unaryExpression| '--' unaryExpression| 'sizeof' unaryExpression`等规则来说，它却是有意义的——因为它携带了符号等信息。因此我们使用别名`unaryExpression_`，并正确构造它们的`enter`和`exit`方法即可。

方法的容错性/可拓展性

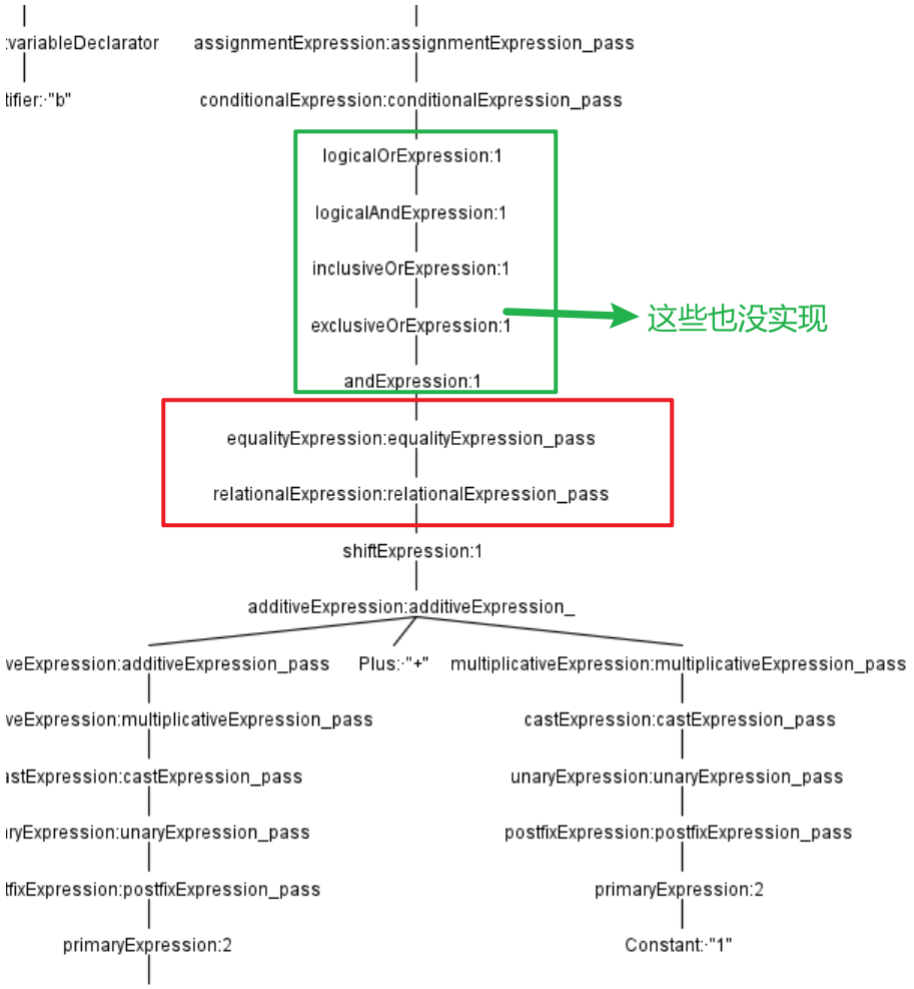
本方法最大的亮点，就是在于容错和可拓展性上了。

我之所以觉得这个方法十分优雅的原因，就在于它在保持简单且正确的同时，有着非常强大的容错性和可拓展性。

对于容错性，因为算法保证每个子节点**只向前看一个节点**，因此保证了不会有堆栈平衡上的问题。并且，子节点在挂载前会先判断父节点，并且退出时会将自己弹出，这就使得堆栈能从始至终一直保持相对正确——即使子节点判断父节点失败（比如法则的缺失），也并不会污染堆栈（子节点不会留在栈中）；同时由于判断了栈顶元素，所以确保了挂载属性的正确性，在确定栈顶元素的情况下，可以极其方便的使用面向对象的**多态性**，并且不会出现盲目的向上/向下造型。这种方法在**最坏情况下，是漏挂节点**，而不会出现耦合型、递归型甚至Java语法层面的错误。

对于可拓展性，最直观表现的就是对于表达式的嵌套上。由于本方法允许不同节点之间“有意义”的节点在语法树上的对应区域距离很远，并且节点的连接是自动完成的，因此就可以保证即使中途**引入新节点，也不需要修改已经写好的代码。**

例如在上述表达式中，在一开始的过程中给你`equalityExpression`方法并没有实现：



在后续对它有添加的必要后，我只在源代码的基础上新增了个位数行的代码：

```
//visitTerminal switch case处添加一条判断（1行代码）
...
case CLexer.Greater:
...

@Override
public void enterEqualityExpression_(EqualityExpression_Context ctx) {
    nodeStack.push(new ASTBinaryExpression());//节点压栈，1行代码
}

@Override
public void exitEqualityExpression_(EqualityExpression_Context ctx) {
    exitExpressionNode();//节点退栈，1行代码
}
```

实际自己新增的代码只需要3行，便能完成对一个新的表达式节点的增加。

倘若使用`visitor`的方法，或许还需要考虑新老节点之间的传递和遍历关系。

事实上，上图绿框处也是目前还没实现的语法规则，若后续有必要实现，也仅仅需要略微新增加几行代码即可。

由于本方法的简洁、正确、强大的可拓展性（优雅），使得最后在编程过程中，对于语法树的遍历、“模拟”AST的过程变为了一种纯粹的硬编码——只需要不停的`push`、`pop`、挂载就可以扩展语法规则，而无需进一步考虑细节。并且由于前文所说的容错性，**即使省略了许多语法节点和规则不写**，最坏情况也只是**漏挂**节点，而树本身仍然可以正常遍历和运行，使得整体代码一直是处在“可以跑”的状态，可以很大程度上增加开发效率。

在最后代码完成时，在**基本实现C语言的所有语法特性**的情况下，使用标准的`JAVA`格式格式化后，代码行数不到600行。在考虑补全、换行符等因素下，自己需要完成的编码远少于这个数。

```
559     }
560
561     @Override
562     public void exitArrayAceess_(ArrayAceess_Context ctx) {
563         exitExpressionNode();
564     }
565
566 }
567
```

开发效率作为对比：使用本方法完成实验，完成编码和测试花费了1.5天的时间。（对比身边同学的3天以上）

参考资料

> 冯开宇学长的antlr讲解视频
>
> Antlr文件书写：antlr/grammars-v4: Grammars written for ANTLR v4; expectation that the grammars are free of actions. (github.com)
>
> CST和AST：parsing - What is the difference between an Abstract Syntax Tree and a Concrete Syntax Tree? - Stack Overflow
>
> 文章所涉及的项目：jiweixing/BIT-MiniCC: A C compiler framework in Java (github.com)