词法分析

Hollow Man

实验原理分析

定义：

语法分析器这里使用递归下降方法。其为每个非终结符编制一个递归下降分析函数，每个函数名是相应的非终结符，函数体则是根据规则右部符号串的结构和顺序编写。随后子程序相互递归调用。

输入：

词法分析器所输出单词符号输出成如下的二元式：

（单词内容，单词种别）

输出：

一个 XML 文件，定义了语法树的结构。随后使用 python treelib 库生成 dot 文件，使用其通过 graphviz 生成 png 语法树图片。

算法流程分析

这里使用的文法如下：

Block -> { structDeclar | functionDeclar }

structDeclar -> struct identifier { {memberDeclar} }

memberDeclar -> varDeclar

varDeclar -> type identifier {, identifier} ;

type -> int | char | boolean | identifier

functionDeclar -> (type|void) identifier (paramList) functionBody

paramList -> type identifier {, type identifier} | ε

functionBody -> { {statement} }

statement -> varDeclarStatement | letStatemnt | ifStatement | whileStatement | doStatement | returnStatemnt | functionCall

varDeclarStatement -> var type identifier { , identifier } ;

letStatemnt -> let identifier [[expression]] = expression ;

ifStatement -> if (expression) { {statement} } [else { {statement} }]

whileStatement -> while (expression) { {statement} }

doStatement -> do { statement }

functionCall -> identifier [. identifier] (expressionList);

expressionList -> expression { , expression } | ε

returnStatemnt -> return [expression] ;

expression -> relationalExpression { (& | |) relationalExpression }

relationalExpression -> ArithmeticExpression { (= | > | <) ArithmeticExpression }

ArithmeticExpression -> term { (+ | -) term }

term -> factor { (* | /) factor }

factor -> (- | ~ | ε) operand

operand -> integerConstant | identifier [.identifier] [[expression] | (expressionList)] | (expression) | stringLiteral | true | false | null | this

程序关键部分分析

1. **关键部分分析**

使用递归下降文法，*Test()函数来进行 First 集选择路径，PeekNextToken()方法来使用 Follow 集。在 if..else..中插入错误判断以及 XML 语法树生成。如遇到错误程序立刻停止不再分析。

2. **结果分析**

使用了以下程序测试文件：

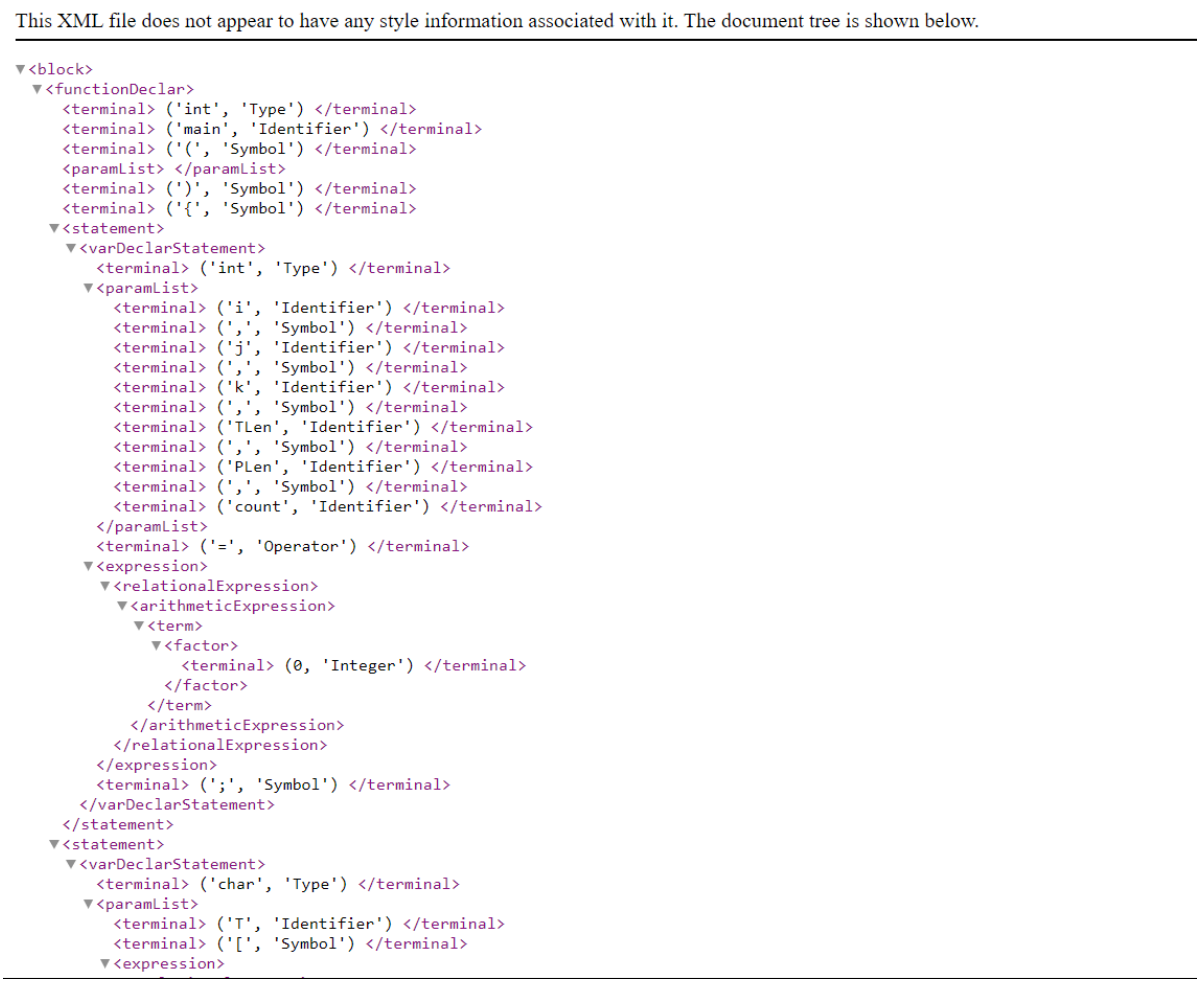
```
int main()
{
    //sdfdsfdfsdsfsdf
  int i,j,k,TLen,PLen,count=0;
  char T[50],P[10];
  TLen=T;
    int a;
```

```
    int b;
 char k[3]="a\"bc";

/*
sdfsdfdfs
*/

    a = 1;
 a=a+1;
    b = a + 2 ;
 PLen=P;
 while(i<TLen-PLen)
 {
    if(a&b){
        i=i+1;
    }
    else{
    b=a;
    }

    if(b){
        i=(i+1)*2;
    }

 }
 return 0;
```

}

程序能够正常输出词法分析运行结果，并且无语法报错情况发生。在上程序中故意引入错误，程序能够正常识别出错误类型。

生成 XML 文件样例如下：



This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
▼<block>
  ▼<functionDeclar>
     <terminal> ('int', 'Type') </terminal>
     <terminal> ('main', 'Identifier') </terminal>
     <terminal> ('(', 'Symbol') </terminal>
     <paramList> </paramList>
     <terminal> (')', 'Symbol') </terminal>
     <terminal> ('{', 'Symbol') </terminal>
   ▼<statement>
    ▼<varDeclarStatement>
       <terminal> ('int', 'Type') </terminal>
      ▼<paramList>
         <terminal> ('i', 'Identifier') </terminal>
         <terminal> (',', 'Symbol') </terminal>
         <terminal> ('j', 'Identifier') </terminal>
         <terminal> (',', 'Symbol') </terminal>
         <terminal> ('k', 'Identifier') </terminal>
         <terminal> (',', 'Symbol') </terminal>
         <terminal> ('TLen', 'Identifier') </terminal>
         <terminal> (',', 'Symbol') </terminal>
         <terminal> ('PLen', 'Identifier') </terminal>
         <terminal> (',', 'Symbol') </terminal>
         <terminal> ('count', 'Identifier') </terminal>
       </paramList>
       <terminal> ('=', 'Operator') </terminal>
      ▼<expression>
       ▼<relationalExpression>
        ▼<arithmeticExpression>
         ▼<term>
          ▼<factor>
             <terminal> (0, 'Integer') </terminal>
           </factor>
         </term>
        </arithmeticExpression>
       </relationalExpression>
      </expression>
       <terminal> (';', 'Symbol') </terminal>
    </varDeclarStatement>
   </statement>
  ▼<statement>
    ▼<varDeclarStatement>
       <terminal> ('char', 'Type') </terminal>
      ▼<paramList>
         <terminal> ('T', 'Identifier') </terminal>
         <terminal> ('[', 'Symbol') </terminal>
       ▼<expression>
```

随后进行 xml 文件的解析，使用 treelib，将树存储在数据结构中，并打印树的形状：

block

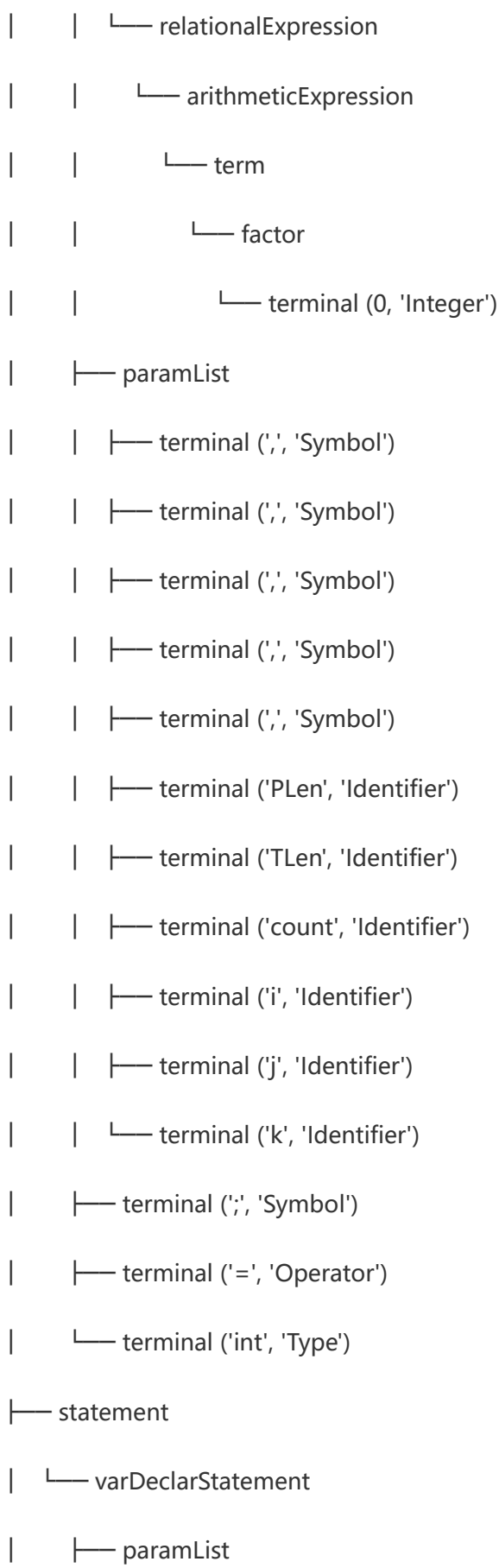
└── functionDeclar

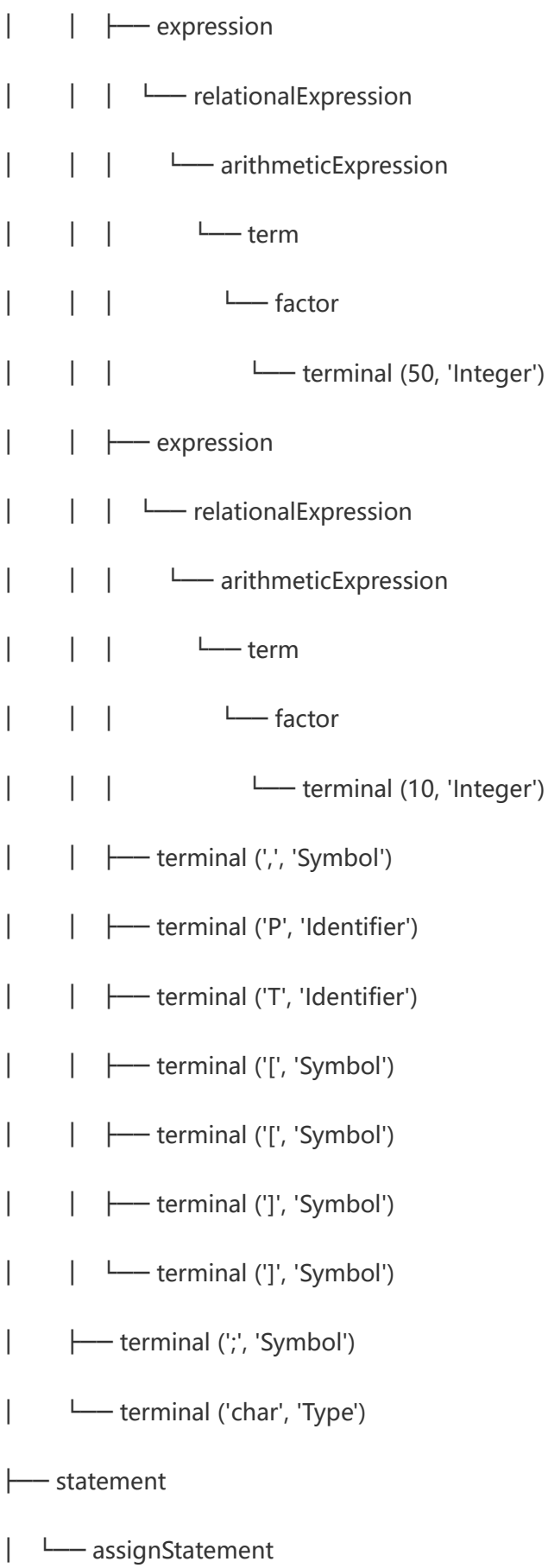
    ├── paramList

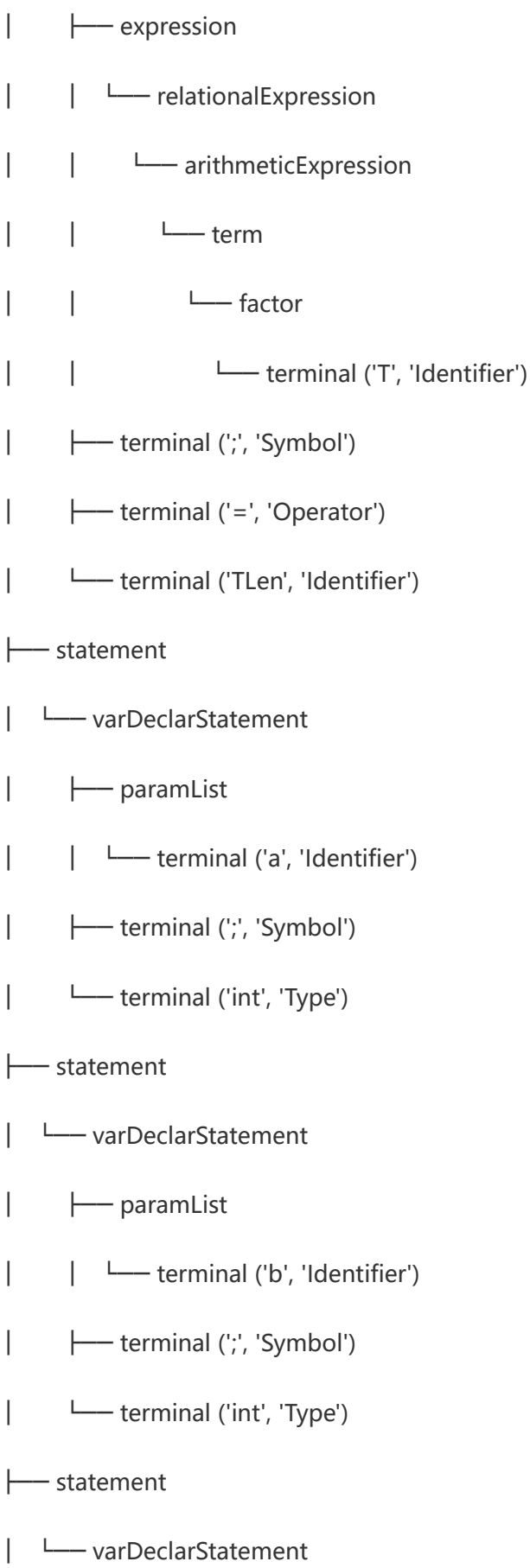
    ├── statement

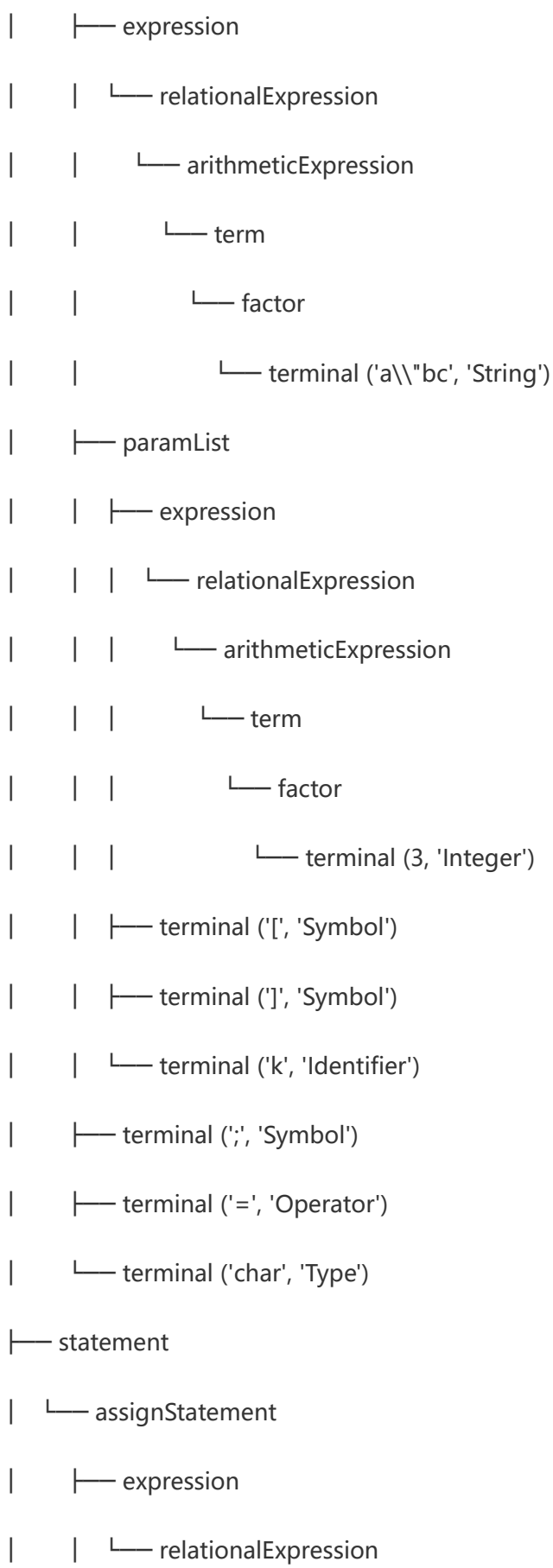
    | └── varDeclarStatement

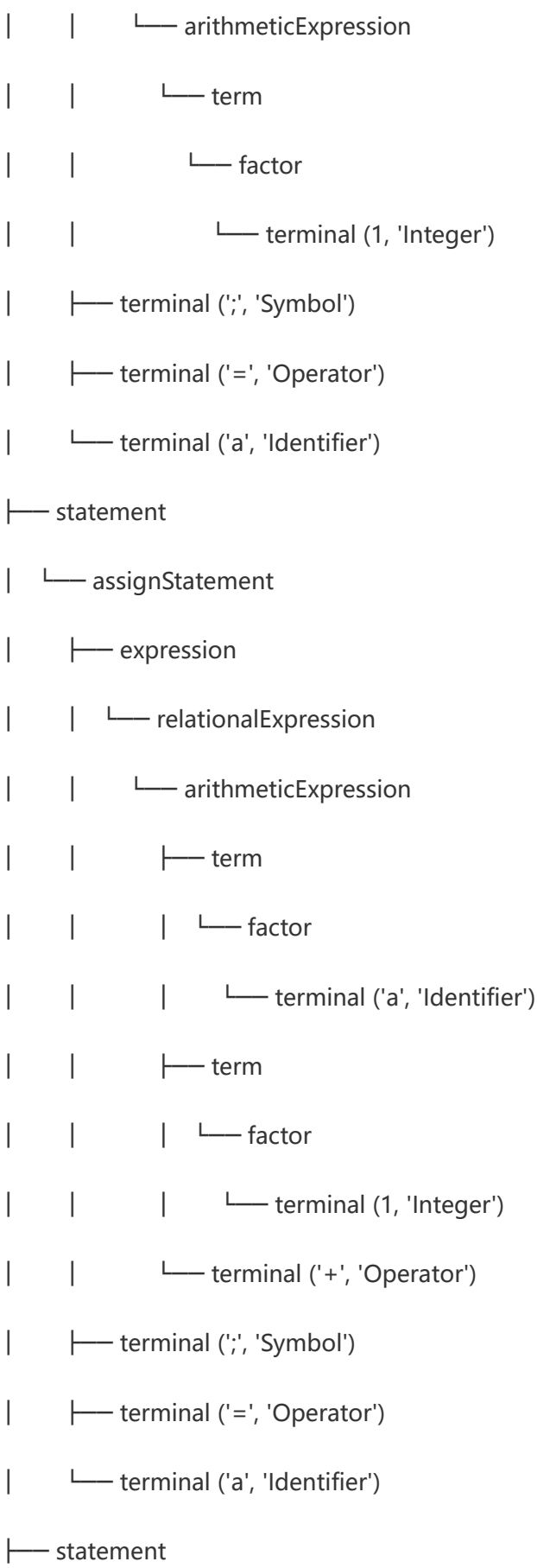
    | ├── expression

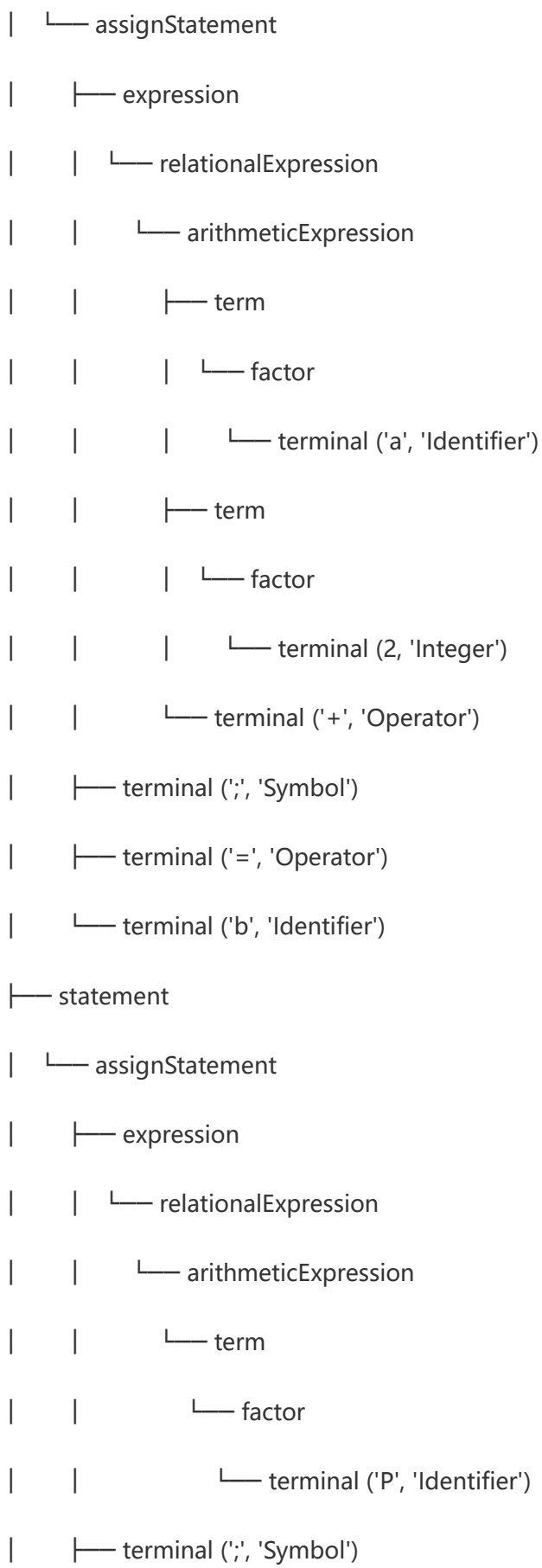

relationalExpression
arithmeticExpression
term
factor
terminal (0, 'Integer')
paramList
terminal (',', 'Symbol')
terminal (',', 'Symbol')
terminal (',', 'Symbol')
terminal (',', 'Symbol')
terminal (',', 'Symbol')
terminal ('PLen', 'Identifier')
terminal ('TLen', 'Identifier')
terminal ('count', 'Identifier')
terminal ('i', 'Identifier')
terminal ('j', 'Identifier')
terminal ('k', 'Identifier')
terminal (';', 'Symbol')
terminal ('=', 'Operator')
terminal ('int', 'Type')
statement
varDeclarStatement
paramList

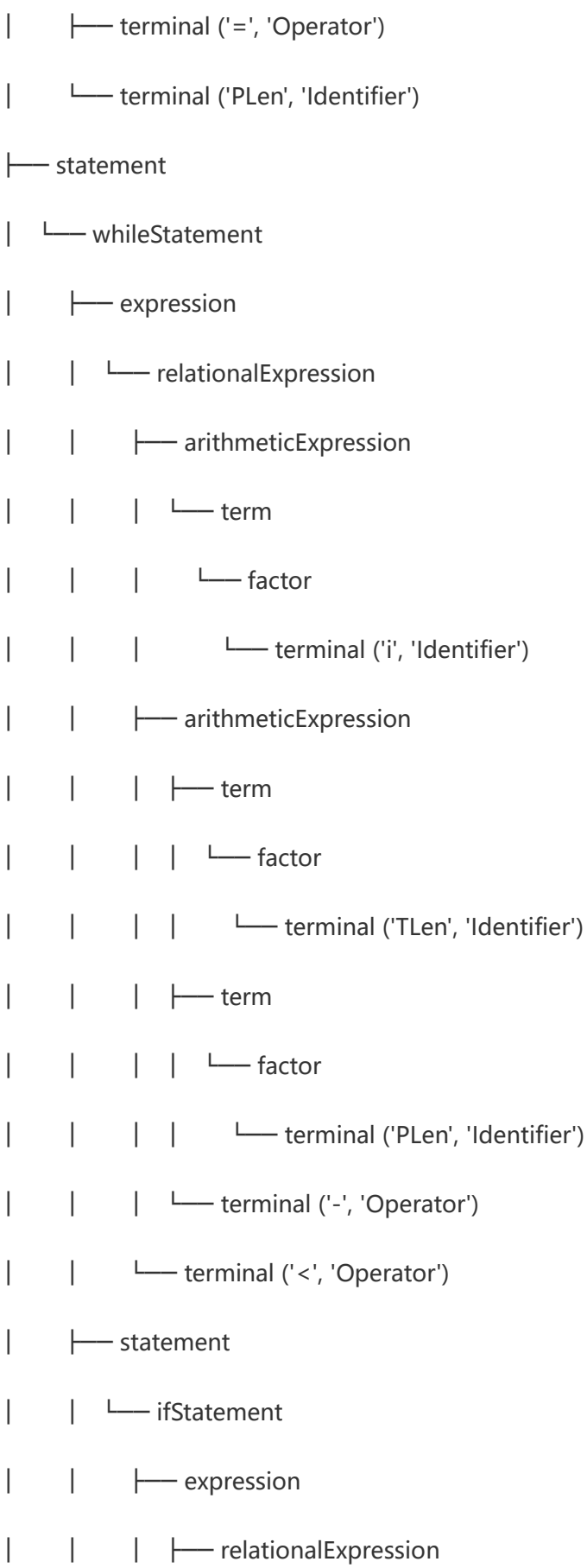

expression
relationalExpression
arithmeticExpression
term
factor
terminal (50, 'Integer')
expression
relationalExpression
arithmeticExpression
term
factor
terminal (10, 'Integer')
terminal (',', 'Symbol')
terminal ('P', 'Identifier')
terminal ('T', 'Identifier')
terminal ('[', 'Symbol')
terminal ('[', 'Symbol')
terminal (']', 'Symbol')
terminal (']', 'Symbol')
terminal (';', 'Symbol')
terminal ('char', 'Type')
statement
assignStatement

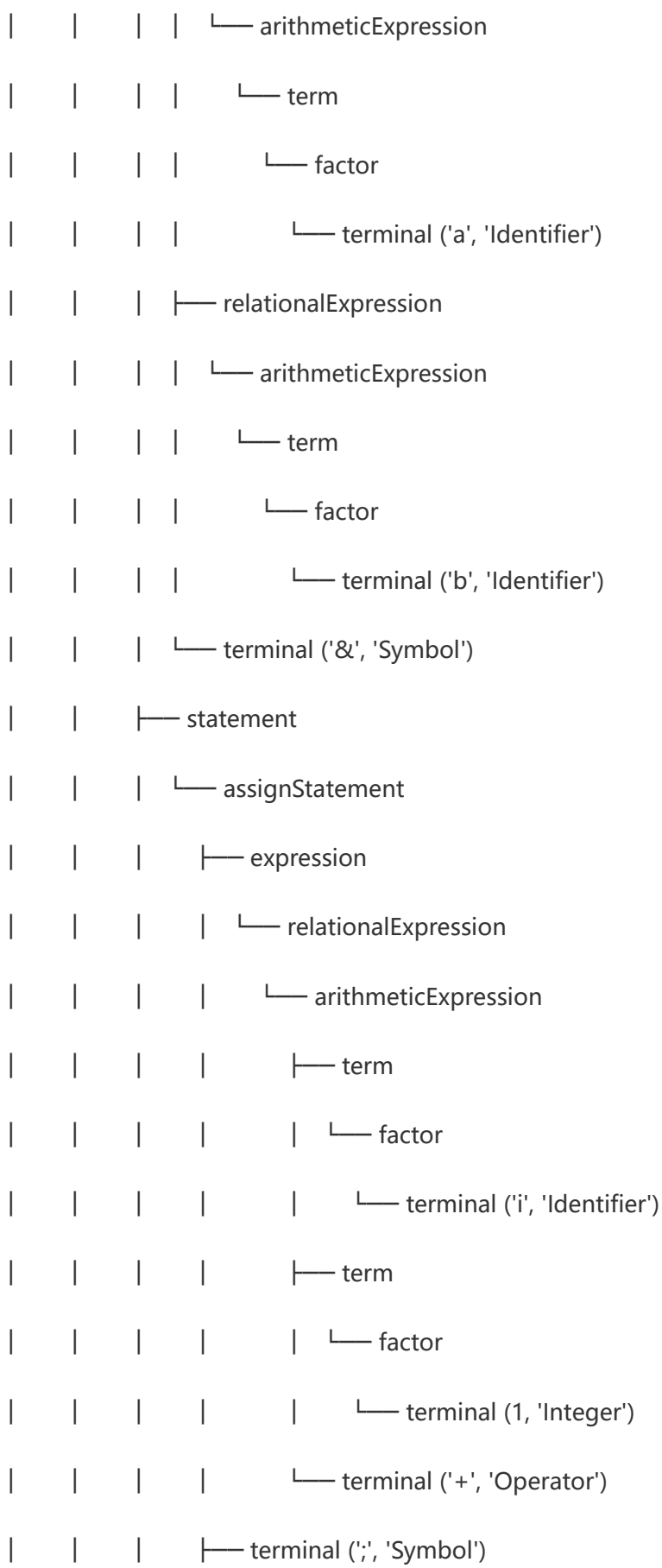

expression
relationalExpression
arithmeticExpression
term
factor
terminal ('T', 'Identifier')
terminal (';', 'Symbol')
terminal ('=', 'Operator')
terminal ('TLen', 'Identifier')
statement
varDeclarStatement
paramList
terminal ('a', 'Identifier')
terminal (';', 'Symbol')
terminal ('int', 'Type')
statement
varDeclarStatement
paramList
terminal ('b', 'Identifier')
terminal (';', 'Symbol')
terminal ('int', 'Type')
statement
varDeclarStatement

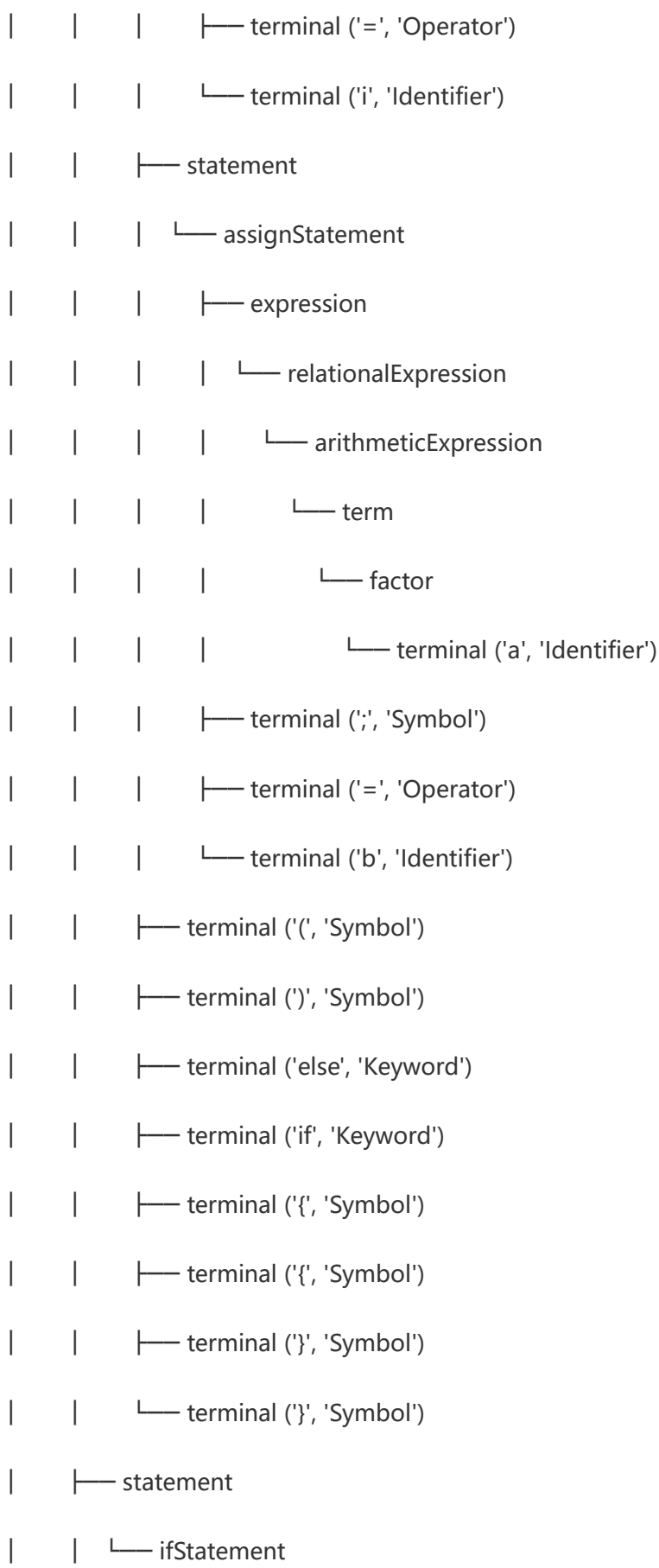

expression
relationalExpression
arithmeticExpression
term
factor
terminal ('a\\"bc', 'String')
paramList
expression
relationalExpression
arithmeticExpression
term
factor
terminal (3, 'Integer')
terminal ('[', 'Symbol')
terminal (']', 'Symbol')
terminal ('k', 'Identifier')
terminal (';', 'Symbol')
terminal ('=', 'Operator')
terminal ('char', 'Type')
statement
assignStatement
expression
relationalExpression

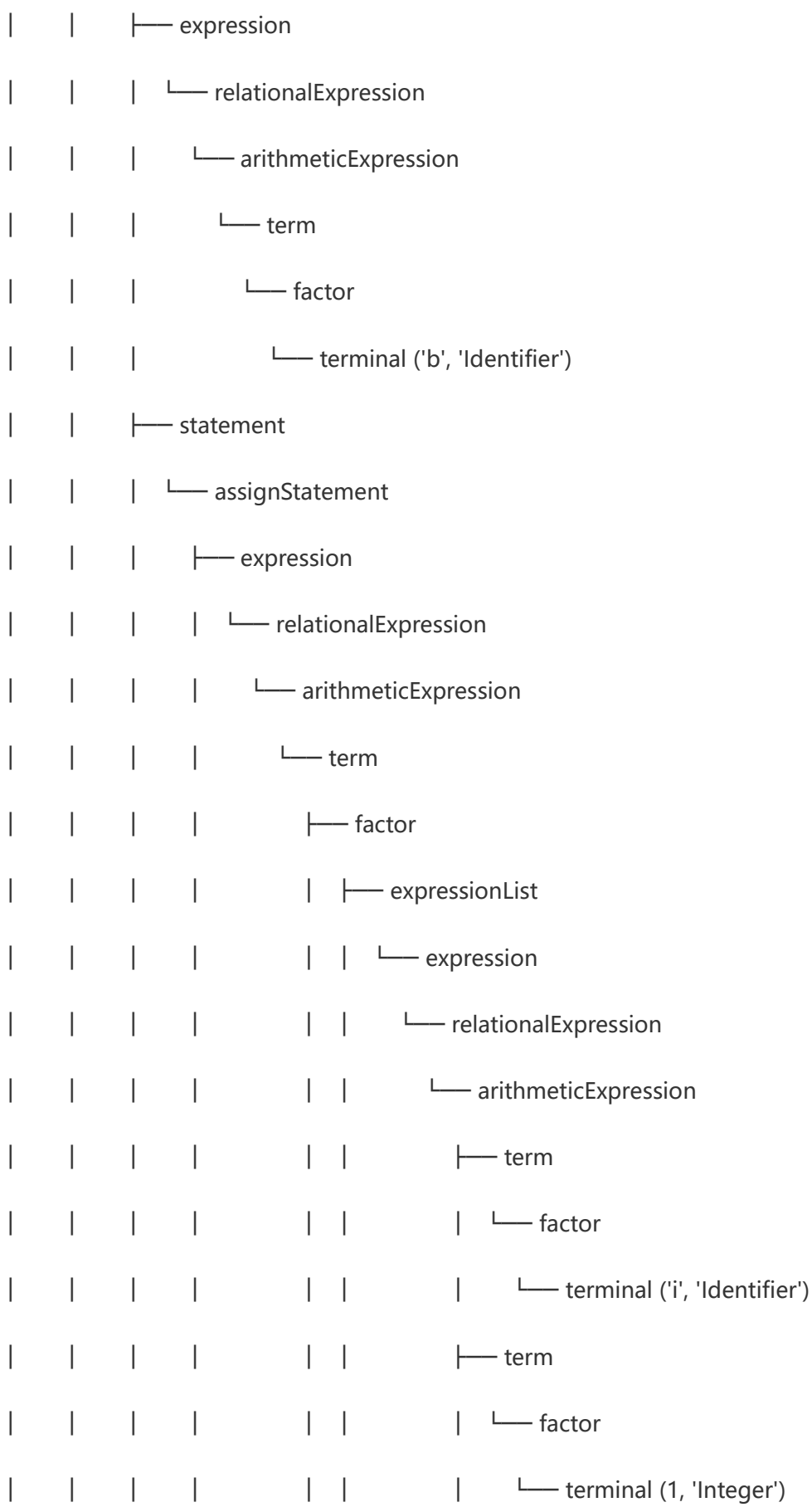

└── arithmeticExpression
└── term
└── factor
└── terminal (1, 'Integer')
├── terminal (';', 'Symbol')
├── terminal ('=', 'Operator')
└── terminal ('a', 'Identifier')
├── statement
└── assignStatement
├── expression
└── relationalExpression
└── arithmeticExpression
├── term
└── factor
└── terminal ('a', 'Identifier')
├── term
└── factor
└── terminal (1, 'Integer')
└── terminal ('+', 'Operator')
├── terminal (';', 'Symbol')
├── terminal ('=', 'Operator')
└── terminal ('a', 'Identifier')
├── statement

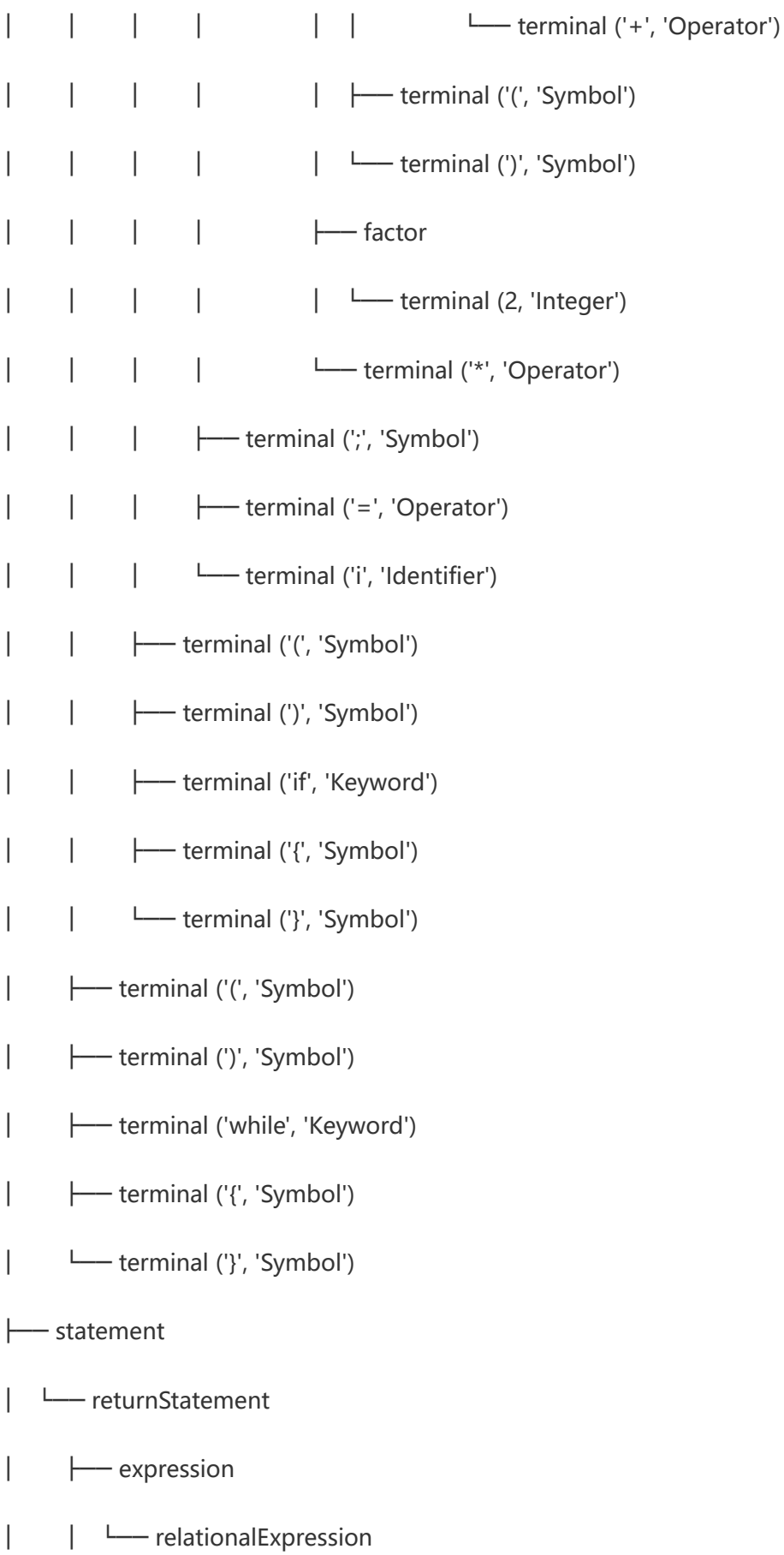

└── assignStatement
├── expression
│ └── relationalExpression
│ └── arithmeticExpression
│ ├── term
│ │ └── factor
│ │ └── terminal ('a', 'Identifier')
│ ├── term
│ │ └── factor
│ │ └── terminal (2, 'Integer')
│ └── terminal ('+', 'Operator')
├── terminal (';', 'Symbol')
├── terminal ('=', 'Operator')
└── terminal ('b', 'Identifier')
├── statement
└── assignStatement
├── expression
│ └── relationalExpression
│ └── arithmeticExpression
│ └── term
│ └── factor
│ └── terminal ('P', 'Identifier')
├── terminal (';', 'Symbol')


terminal ('=', 'Operator')
terminal ('PLen', 'Identifier')
statement
whileStatement
expression
relationalExpression
arithmeticExpression
term
factor
terminal ('i', 'Identifier')
arithmeticExpression
term
factor
terminal ('TLen', 'Identifier')
term
factor
terminal ('PLen', 'Identifier')
terminal ('-', 'Operator')
terminal ('<', 'Operator')
statement
ifStatement
expression
relationalExpression


└── arithmeticExpression
└── term
└── factor
└── terminal ('a', 'Identifier')
├── relationalExpression
└── arithmeticExpression
└── term
└── factor
└── terminal ('b', 'Identifier')
└── terminal ('&', 'Symbol')
├── statement
└── assignStatement
├── expression
└── relationalExpression
└── arithmeticExpression
├── term
└── factor
└── terminal ('i', 'Identifier')
├── term
└── factor
└── terminal (1, 'Integer')
└── terminal ('+', 'Operator')
├── terminal (';', 'Symbol')


terminal ('=', 'Operator')
terminal ('i', 'Identifier')
statement
assignStatement
expression
relationalExpression
arithmeticExpression
term
factor
terminal ('a', 'Identifier')
terminal (';', 'Symbol')
terminal ('=', 'Operator')
terminal ('b', 'Identifier')
terminal ('(', 'Symbol')
terminal (')', 'Symbol')
terminal ('else', 'Keyword')
terminal ('if', 'Keyword')
terminal ('{', 'Symbol')
terminal ('{', 'Symbol')
terminal ('}', 'Symbol')
terminal ('}', 'Symbol')
statement
ifStatement


expression
relationalExpression
arithmeticExpression
term
factor
terminal ('b', 'Identifier')
statement
assignStatement
expression
relationalExpression
arithmeticExpression
term
factor
expressionList
expression
relationalExpression
arithmeticExpression
term
factor
terminal ('i', 'Identifier')
term
factor
terminal (1, 'Integer')


└── terminal ('+', 'Operator')
├── terminal ('(', 'Symbol')
└── terminal (')', 'Symbol')
├── factor
└── terminal (2, 'Integer')
└── terminal ('*', 'Operator')
├── terminal (';', 'Symbol')
├── terminal ('=', 'Operator')
└── terminal ('i', 'Identifier')
├── terminal ('(', 'Symbol')
├── terminal (')', 'Symbol')
├── terminal ('if', 'Keyword')
├── terminal ('{', 'Symbol')
└── terminal ('}', 'Symbol')
├── terminal ('(', 'Symbol')
├── terminal (')', 'Symbol')
├── terminal ('while', 'Keyword')
├── terminal ('{', 'Symbol')
└── terminal ('}', 'Symbol')
├── statement
└── returnStatement
├── expression
└── relationalExpression

```
    |       |       └── arithmeticExpression
    |       |           └── term
    |       |               └── factor
    |       |                   └── terminal (0, 'Integer')
    |       ├── terminal (';', 'Symbol')
    |       └── terminal ('return', 'Keyword')
    ├── terminal ('(', 'Symbol')
    ├── terminal (')', 'Symbol')
    ├── terminal ('int', 'Type')
    ├── terminal ('main', 'Identifier')
    ├── terminal ('{', 'Symbol')
    └── terminal ('}', 'Symbol')
```

再调用该库导出生成 dot 文件，使用 graphviz 进行可视化，得到如下结果：

实验总结

1. **编译环境**

Fedora 33

2. **语言环境**

Python 3.9