華南師範大學

本科学生实验（实践）报告

院　　系：　计算机学院

实验课程：编译原理

实验项目：实验 3

指导老师：黄煜廉

开课时间：2023～2024 年度第二学期

专　　业：计算机科学与技术

班　　级：计科 2 班

华南师范大学教务处

华 南 师 范 大 学 实 验 报 告

学生姓名 林泽勋 **学　　号** 20212821020

专　　业 计算机科学与技术 **年级班级** 22 级计科 2 班

课程名称 编译原理 **实验项目** 实验 3

实验类型 □验证 □设计 □综合 **实验时间** 2024 年 5 月 8 日

实验指导老师 黄煜廉 **实验评分**

一、实验内容

（一）为 Tiny 语言扩充的语法有

1. 实现改写书写格式的新 if 语句；

2. 扩充算术表达式的运算符号：++（前置自增 1）、 --（前置自减 1）运算符号（类似于 C 语言的++和--运算符号，但不需要完成后置的自增 1 和自减 1）、求余%、乘方^;

3. 扩充扩充比较运算符号：<(小于)、>(大于)、<=(小于等于)、>=(大于等于)、<>(不等于)等运算符号;

4. 增加正则表达式，其支持的运算符号有： 或(|) 、连接(&)、闭包(#)、括号( ) 、可选运算符号（？）和基本正则表达式。

5. for 语句的语法规则（类似于 C 语言的 for 语言格式）：书写格式：for(循环变量赋初值;条件;循环变量自增或自减 1) 语句序列

6. while 语句的语法规则（类似于 C 语言的 while 语言格式）：书写格式：while(条件) 语句序列 endwhile

（二）对应的语法规则分别为：

1. 把 TINY 语言原有的 if 语句书写格式

if_stmt-->if exp then stmt-sequence end | | if exp then stmt-sequence else stmt-sequence end 改写为：if_stmt-->if(exp) stmt-sequence else stmt-sequence | if(exp) stmt-sequence

2. ++（前置自增 1）、 --（前置自减 1）运算符号、求余%、乘方^等运算符号的文法规则请自行组织。

3. <(小于)，>(大于)、<=(小于等于)、>=(大于等于)、<>(不等于)等运算符号的文法规则请自行组织。

4. 为 tiny 语言增加一种新的表达式——正则表达式，其支持的运算符号有： 或(|) 、连接(&)、闭包(#)、括号( ) 、可选运算符号（？）和基本正则表达式，对应的文法规则请自行组织。

5. 为 tiny 语言增加一种新的语句，ID==正则表达式 （同时增加正则表达式的赋值运算符号==）

6. 为 tiny 语言增加一个符合上述 for 循环的书写格式的文法规则，

7. 为 tiny 语言增加一个符合上述 while 循环的书写格式的文法规则，

8. 为了实现以上的扩充或改写功能，还需要注意对原 tiny 语言的文法规则做一些相应的改造处理。

Tiny 语言原来的文法规则，可参考：云盘中参考书 P97 及 P136 的文法规则。

要求：

（1）要提供一个源程序编辑的界面，以让用户输入源程序（可输入，可保存、可打开源程序）

（2）可由用户选择是否生成语法树，并可查看所生成的语法树。

（3）实验 3 的实现只能选用的程序设计语言为：C++

（4）要求应用程序的操作界面应为 Windows 界面。

（5）应该书写完善的软件文档

二、实验目的

1. 理解并掌握编程语言的词法分析与语法分析的基本概念和方法：通过实际操作，深入理解词法分析器和语法分析器的工作原理和实现过程。

2. 熟悉语言扩充的流程：学习如何在现有编程语言的基础上进行语法扩充，增强语言的表达能力和灵活性。

3. 实践编程语言的语法规则设计：通过自行设计语法规则，加深对编程语言设计原则的理解。

4. 提高编程能力：通过编写词法分析器和语法分析器，提升编程实践能力，

尤其是对复杂逻辑的实现能力。

5. 增强问题解决能力：面对语言扩充中遇到的各种问题，学会如何分析问题并找到解决方案。

三、实验文档：

1　系统总体设计

1.1 Tiny 语言介绍

Tiny 语言的语法为：

- 注释：放在一对大括号内，不能嵌套；
- 关键字：read write if end repeat until else；
- 类型：只支持整型和布尔型；
- 运算符：+ - * / ( ) < = :=，其中:=为赋值运算，=为判断。没有>和<=和>=。

这个语法并不能完全适配这次实验：在本次实验中对文法与符号进行扩充，如：新的运算符、正则表达式、正则表达式新的基本单位——字符类型等。并扩充了浮点数类型等拓展内容（Tiny 语言仅有整数这一数字类型）。

1.2 总体流程图

本程序从 Tiny 语言出发，通过分析待扩充的语法，写出扩充后的文法规则，根据文法规则扩充新的 tiny 语言符号。进而进行 Tiny 语言的词法分析，并在去除注释类型 token 后，对 Tiny 进行语法分析。最后，对词法分析与语法分析结果进行展示。

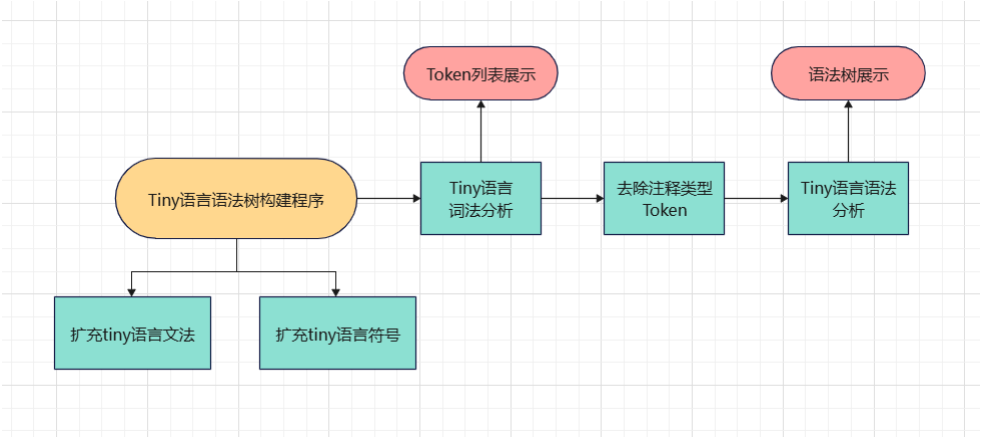

图 1 总体流程图

2 Tiny 语言的词法分析实现

2.1 扩充的类型

- 数字类型扩充——浮点数：如-1.2e10、2.3E3.2 等，均视为浮点数（数字类型的一种）
- 字符：以单引号包裹的字符，如：’a’,‘b’,’c’等，作为正则表达式的基本单元。

2.2 词法分析流程

2.2.1 Token 数据结构

```
/**
 * @brief The LexerInfo struct
 * 记录分词信息
 * 类型：
 * 0：分隔符号
 * 1：字符
 * 2：数字
 * 3：标识符
 * 4：关键字
 * 5：运算符
 * 6：注释
 */
struct LexerInfo {
    int lexerType;  // 分词类别
    int row;        // 行
    int column;     // 列
    LexerInfo(int type, int r, int c): lexerType(type), row(r), column(c) {}
    LexerInfo(): LexerInfo(0, 0, 0) {}
};
```

2.2.2 分词流程

- 读取关键字与运算符的映射：
  - 使用 QFile 读取一个名为 mapping.json 的文件，该文件可能包含了关键字、运算符与它们对应的分类（如关键字、分隔符、运算符等）。使用 QJsonDocument 解析 JSON 数据，并将其存储在 QHash<QString, QString> 中，其中 key 是字符或字符串，value 是它们的分类。

- 词法分析主过程：
  - 初始化一个空的 tokenList 用于存储分析结果。使用 split 函数按行分割输入内容。遍历每一行，对每一行进行词法分析，包括跳过空白字符、识别分隔符号、字符类型、数字类型、标识符或关键字、注释和运算符。
  - 处理各种词法单元：
    - 分隔符号：如遇 (、)、; 等，直接添加到 tokenList。
    - 字符类型：识别单引号包围的字符，并添加到 tokenList。
    - 数字类型：识别数字（可能包含小数点和科学计数法），并添加到 tokenList。
    - 标识符或关键字：识别字母或下划线开头的字符串，根据 hash 判断是关键字还是标识符，并添加到 tokenList。
    - 注释：识别以 { 开头的注释，并找到对应闭合的 }，将整个注释块作为一个词法单元添加到 tokenList。
    - 运算符：识别单个字符或双字符的运算符，并添加到 tokenList。
- 错误处理：
  - 如果在字符类型中未找到闭合的单引号，或在注释中未找到闭合的 }，则跳出当前循环。
- 返回结果：
  - 将填充好的 tokenList 返回。

具体分词代码见附录。

3 Tiny 语言的语法分析实现

3.1 改进后语法错误分析

按照改进后的 if 语句进行编写时，笔者发现了 if 语句最后的分号归属问题。由于没有 endif 等分隔符，文法出现了二义性。如：

```
read x;
```

```
if x = 1:
    x := 2;
write x
```

这段程序中，x:=2 后的分号，并不能被准确的判断。

可能性一：分号是 if 中的 stmt-sequence 的分号，这种情况下，write x 会被当做 if 作用域内的语句。

可能性二：分号是整个 program 中的 stmt-sequence 的分号，这种情况下，分号被视作 if 结束的标志，write x 是在 if 作用域外的。

上述错误会导致文法产生二义性，应当改正。在本实验中，笔者将在 if 语句最后加上 endif；for 语句最后加上 endfor；while 语句最后加上 endwhile，以消除这种二义性错误。具体文法可见 3.2 展示。

3.2 扩充后的 Tiny 语言文法

图 3 至图 5 展示了扩充后的 Tiny 语言文法，其中用黄色标识的是终结符。

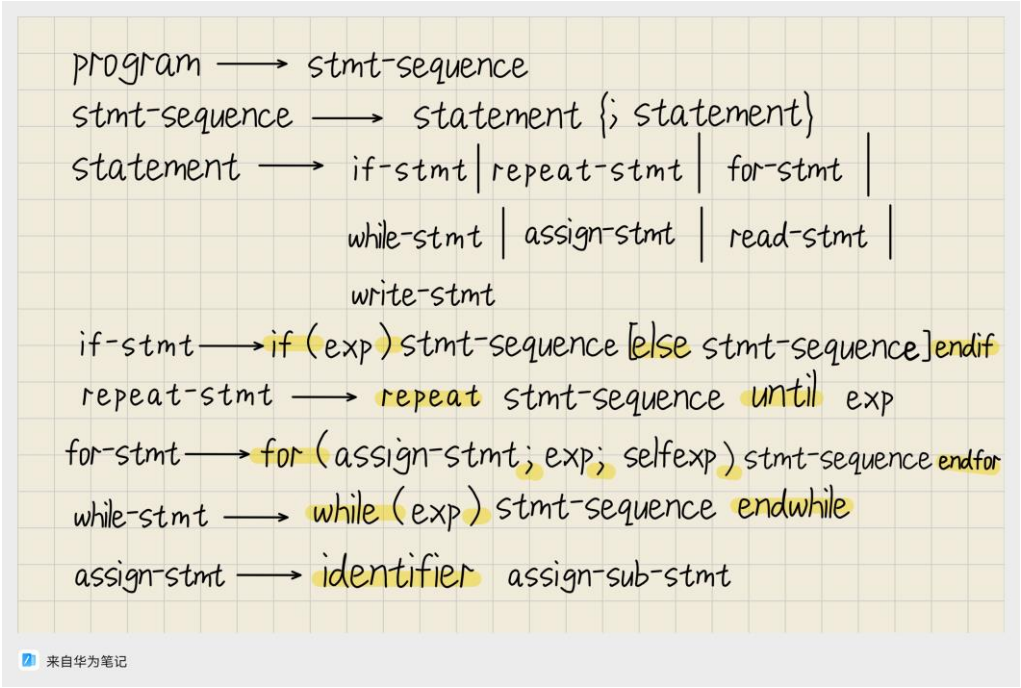

图 2 扩充 Tiny 语言完整文法-1

read-stmt ⟶ read identifier

write-stmt ⟶ write write-sub-stmt

write-sub-stmt ⟶ exp | letter

assign-sub-stmt ⟶ := exp | == re

re ⟶ orre { | orre }

orre ⟶ conre { & conre }

conre ⟶ repre [repop]

repop ⟶ # | ?

repre ⟶ (re) | letter | identifier

exp ⟶ simple-exp [comparison-op simple-exp]

来自华为笔记

图 3 扩充 Tiny 语言完整文法-2

comparison-op ⟶ < | = | > | <= | <> | =>

simple-exp ⟶ term {addop term}

addop ⟶ + | -

term ⟶ factor {mulop factor}

mulop ⟶ * | / | %

factor ⟶ resexp { ^ resexp }

resexp ⟶ (exp) | number | identifier | selfexp

selfexp ⟶ selfop identifier

selfop ⟶ ++ | --

来自华为笔记

图 4 扩充 Tiny 语言完整文法-3

修改或扩展 if 语句、for 语句、while 语句文法：

if-stmt --> if (exp) stmt-sequence [else stmt-sequence] endif

for-stmt --> for (assign-stmt; exp; selfexp) stmt-sequence endfor

while-stmt --> while (exp) stmt-sequence endwhile

其中，本程序新增了子赋值语句（assign-sub-stmt）用于判断赋值

语句是算术表达式赋值还是正则表达式赋值。

assign-stmt --> identifier assign-sub-stmt

assign-sub-stmt --> :== exp | == re

扩充算术表达式的%运算、^运算与前置自增自减：

mulop 添加%

selfexp --> selfop identifier

resexp 添加 selfexp

扩充正则表达式运算：

re --> orre { | orre }

orre --> conre { & conre }

conre --> repre [repop]

repre --> (re) | letter | identifier

4 Tiny 语言的文法分析实现

4.1 语法树的数据结构设计

```
1.    typedef struct SyntaxNode {
2.        QString nodeStr;
3.        QVector<SyntaxNode *> children;
4.        QVector<SyntaxNode *> brother;
5.
6.        SyntaxNode(QString str): nodeStr(str), children(QVect
   or<SyntaxNode *>()), brother(QVector<SyntaxNode *>()) {}
7.        SyntaxNode(): SyntaxNode("") {}
8.    }SyntaxNode, * SyntaxTree;
```

这段代码定义了一个名为 SyntaxNode 的 C++结构体，它用于构建一个抽象语法树（Abstract Syntax Tree，AST）。抽象语法树在编译器设计中广泛使用，用于表示源代码的结构。下面是对结构体和其中的方法的介绍：

成员变量：

nodeStr: QString 类型，用于存储当前节点的字符串表示，这可以是关键字、标识符、操作符等。

children: QVector<SyntaxNode *>类型，是一个动态数组，用于存储当前节点的子节点。在 AST 中，每个节点都可以有零个或多个子节

点，这些子节点代表了构成该节点的更小的语法单元。

brother: QVector<SyntaxNode *>类型，也是一个动态数组，用于存储当前节点的兄弟节点。兄弟节点是指具有相同父节点的节点。

构造函数：

SyntaxNode(QString str): 带有一个参数的构造函数，用于创建一个带有特定字符串的 SyntaxNode 对象。这个字符串参数 str 初始化 nodeStr 成员变量。

SyntaxNode(): 无参构造函数，创建一个默认的 SyntaxNode 对象，其 nodeStr 被初始化为空字符串。

4.2 递归子程序实现

4.2.1 getToken、match、ERROR 函数

getToken()

- 功能：获取下一个 token。
- 参数：无。
- 返回值：无。
- 逻辑：

如果 tokenIndex 等于 tokenList 的大小（即已经处理完所有的 tokens），则将 TOKEN 设置为空字符串，curType 设置为 6（可能表示没有更多的 token），并递增 tokenIndex。

如果 tokenIndex 等于 tokenList 大小加 1，表示已经尝试获取过下一个 token，但不存在，此时调用 ERROR(2)记录错误。

否则，将当前 tokenList 中索引为 tokenIndex 的 token 的词面、行号、列号和词法类型分别赋值给 TOKEN、curRow、curCol 和 curType，然后递增 tokenIndex。

match(QString expectedToken)

- 功能：匹配预期的 token 词面。
- 参数：expectedToken：期望匹配的 token 词面。

- 返回值：无。
- 逻辑：

如果当前 TOKEN 与 expectedToken 相等，调用 getToken()获取下一个 token。如果不相等，调用 ERROR(0)记录错误。

match(int expectedTokenType)

- 功能：匹配预期的 token 类型。
- 参数：expectedTokenType：期望匹配的 token 类型。
- 返回值：无。
- 逻辑：

如果当前 curType 与 expectedTokenType 相等，调用 getToken()获取下一个 token。如果不相等，调用 ERROR(1)记录错误。

ERROR(int errorType)

- 功能：记录错误信息。
- 参数：errorType：错误类型，用于确定错误信息的格式。
- 返回值：无。
- 逻辑：

根据 errorType，使用 switch 语句选择相应的错误处理逻辑。每种 case 都会创建一个 ErrorInfo 对象，其中包含错误信息和错误发生的位置（行号和列号），并将这个对象添加到 errorList 中。

```
1.    void MainWindow::getToken()
2.    {
3.        if (tokenIndex == tokenList.size()) {
4.            ++tokenIndex;
5.            TOKEN = "";
6.            curType = 6;
7.            return;
8.        } else if (tokenIndex == tokenList.size() + 1)
9.            ERROR(2);
10.           return;
11.       }
12.       this->TOKEN = tokenList[tokenIndex].first;
13.       this->curRow = tokenList[tokenIndex].second.row
```

```
14.       this->curCol = tokenList[tokenIndex].second.column;
15.       this->curType = tokenList[tokenIndex].second.lexerTyp
    e;
16.       ++tokenIndex;
17.   }
18.
19.   void MainWindow::match(QString expectedToken)
20.   {
21.       if (TOKEN == expectedToken) getToken();
22.       else ERROR(0);
23.   }
24.
25.   void MainWindow::match(int expectedTokenType)
26.   {
27.       if (curType == expectedTokenType) getToken();
28.       else ERROR(1);
29.   }
30.
31.   void MainWindow::ERROR(int errorType)
32.   {
33.       switch (errorType) {
34.       case 0:
35.           errorList.append(ErrorInfo("意外的 token:
    " + TOKEN + "(" + getTypeString(curType) + ")", curRow, curC
    ol));
36.           break;
37.       case 1:
38.           errorList.append(ErrorInfo("意外的 token 类型:
    " + TOKEN + "(" + getTypeString(curType) + ")", curRow, curC
    ol));
39.           break;
40.       case 2:
41.           errorList.append(ErrorInfo("多余的
    token", curRow, curCol));
42.           break;
43.       case 3:
44.           errorList.append(ErrorInfo("存在意外的语句类型
    ", curRow, curCol));
45.           break;
46.       case 4:
47.           errorList.append(ErrorInfo("存在意外的赋值语句
    ", curRow, curCol));
48.           break;
49.       default:
```

```
50.             errorList.append(ErrorInfo("未知错误
   ", curRow, curCol));
51.             break;
52.         }
53.     }
```

4.2.2 program 文法

首先，函数使用 new 关键字动态分配了一个 SyntaxNode 对象，并将其指针赋值给 root。这个节点被构造为包含字符串"start"，表示这是程序的起始节点。随后，函数调用了 stmt_sequence()函数，添加到 root 节点的 children 数组中，作为其子节点。

最后，函数返回 root 指针，这个指针指向整个程序的根节点，连同其子节点（即语句序列）构成了程序的 AST。

```
1.    SyntaxTree MainWindow::program()
2.    {
3.        SyntaxTree root = new SyntaxNode("start");
4.        root->children.append(stmt_sequence());
5.        return root;
6.    }
```

4.2.3 stmt-sequence 文法

首先，函数调用 statement 函数，该函数用来解析单个语句并返回其 AST 节点的指针。接着，函数检查 tmp 是否为 nullptr。如果是，表明 statement 函数没有成功解析出语句，因此整个语句序列也不存在，函数返回 nullptr。函数进入一个循环，条件是 TOKEN 等于分号";"，调用 match 函数，用来判定当前的分号标记，并确认它与预期的分号匹配。再次调用 statement 函数解析下一个语句，并将返回的 AST 节点添加到 tmp 节点的 brother 数组中。这表示这些语句是兄弟节点，都属于同一个语句序列。最后，函数返回 tmp 指针，它指向整个语句序列的 AST 的起始节点。

```
SyntaxTree MainWindow::stmt_sequence()
{
    SyntaxTree tmp = statement();
    if (tmp == nullptr) return nullptr;
    while (TOKEN == ";") {
        match(";");
        tmp->brother.append(statement());
    }
    return tmp;
}
```

4.2.4 statement 文法

函数首先检查当前的语法标记 TOKEN 是否等于字符串"if"。如果是，调用 if_stmt 函数来解析条件语句，并返回其结果。

如果 TOKEN 不是"if"，函数接着检查是否等于"repeat"。如果是，调用 repeat_stmt 函数来解析重复语句。以此类推，函数继续检查 TOKEN 是否等于"for"，如果是，调用 for_stmt 函数来解析循环语句。检查 TOKEN 是否等于"while"，如果是，调用 while_stmt 函数来解析当型循环语句。检查 TOKEN 是否等于"read"，如果是，调用 read_stmt 函数来解析读语句。检查 TOKEN 是否等于"write"，如果是，调用 write_stmt 函数来解析写语句。如果以上条件都不满足，函数检查一个名为 curType 的变量是否等于 3。表示当前标记是一个标识符，如果是，调用 assign_stmt 函数来解析赋值语句。如果所有条件都不满足，函数执行错误处理。ERROR(0);和 ERROR(3);是错误处理函数调用，它们记录错误信息或执行其他错误处理逻辑。然后函数返回 nullptr，表示没有成功解析出语句。

```
SyntaxTree MainWindow::statement()
{
    if (TOKEN == "if") {
        return if_stmt();
    } else if (TOKEN == "repeat") {
        return repeat_stmt();
    } else if (TOKEN == "for") {
        return for_stmt();
    } else if (TOKEN == "while") {
        return while_stmt();
    } else if (TOKEN == "read") {
        return read_stmt();
    } else if (TOKEN == "write") {
        return write_stmt();
    } else if (curType == 3) {
        // 赋值语句：标识符
        return assign_stmt();
    } else {
        ERROR(0);
        ERROR(3);
        return nullptr;
    }
}
```

4.2.5 if-stmt 文法

函数首先调用 match 函数，匹配当前的"if"标记，并确认它与预期的"if"关键字匹配。然后，函数创建一个新的 SyntaxNode 对象，用"if"字符串初始化，这个节点将作为 if 语句的根节点。接着，函数匹配左括号"("，这通常表示 if 语句的条件部分的开始。函数调用 exp 函数来解析条件表达式，并将其返回的 AST 节点添加到 tmp 节点的 children 数组中。函数匹配右括号")"，表示条件部分的结束。函数调用 stmt_sequence 函数来解析 if 语句后面的语句序列，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。函数检查下一个语法标记 TOKEN 是否为"else"。如果是，表示存在一个 else 子句：匹配"else"关键字。调用 stmt_sequence 函数来解析 else 子句中的语句序列，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。最后，函数匹配"endif"标记，这表示 if 语句的结束。函数返回 tmp 指针，它指向表示整个 if 语

句的 AST 的根节点。

```
SyntaxTree MainWindow::if_stmt()
{
    match("if");
    SyntaxTree tmp = new SyntaxNode("if");
    match("(");
    tmp->children.append(exp());
    match(")");
    tmp->children.append(stmt_sequence());
    if (TOKEN == "else") {
        match("else");
        tmp->children.append(stmt_sequence());
    }
    match("endif");
    return tmp;
}
```

4.2.6 repeat-stmt 文法

函数首先调用 match 函数，匹配当前的"repeat"标记，并确认它与预期的"repeat"关键字匹配。然后，函数创建一个新的 SyntaxNode 对象，用"repeat"字符串初始化，这个节点将作为 repeat 语句的根节点。函数调用 stmt_sequence 函数来解析 repeat 循环体中的语句序列，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。接着，函数匹配"until"标记，这通常表示 repeat 循环的条件部分的开始。函数调用 exp 函数来解析循环直到的条件表达式，并将其返回的 AST 节点添加到 tmp 节点的 children 数组中。最后，函数返回 tmp 指针，它指向表示整个 repeat 语句的 AST 的根节点。

```
SyntaxTree MainWindow::repeat_stmt()
{
    match("repeat");
    SyntaxTree tmp = new SyntaxNode("repeat");
    tmp->children.append(stmt_sequence());
    match("until");
    tmp->children.append(exp());
    return tmp;
}
```

4.2.7 for-stmt 文法

函数首先调用 match 函数，匹配当前的"for"标记，并确认它与预期的"for"关键字匹配。然后，函数创建一个新的 SyntaxNode 对象，用"for"字符串初始化，这个节点将作为 for 循环的根节点。接着，函数匹配左括号"("，这表示 for 循环初始化部分的开始。函数调用 assign_stmt 函数来解析赋值语句（通常是循环变量的初始化），并将返回的 AST 节点添加到 tmp 节点的 children 数组中。函数匹配分号";"，表示初始化部分的结束和循环条件的开始。函数调用 exp 函数来解析循环条件表达式，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。函数再次匹配分号";"，表示循环条件的结束和迭代表达式的开始。函数调用 selfexp 函数来解析迭代表达式（通常是循环变量的更新），并将返回的 AST 节点存储在 selfexpTree 指针中。函数匹配右括号")"，表示 for 循环控制部分的结束。函数调用 stmt_sequence 函数来解析 for 循环体中的语句序列，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。将迭代表达式的 AST 节点 selfexpTree 添加到 tmp 节点的 children 数组中。最后，函数匹配"endfor"标记，这表示 for 循环的结束。函数返回 tmp 指针，它指向表示整个 for 循环的 AST 的根节点。

```
1.    SyntaxTree MainWindow::for_stmt()
2.    {
3.        match("for");
4.        SyntaxTree tmp = new SyntaxNode("for");
5.        match("(");
6.        tmp->children.append(assign_stmt());
7.        match(";");
8.        tmp->children.append(exp());
9.        match(";");
10.       SyntaxTree selfexpTree = selfexp();
11.       match(")");
12.       tmp->children.append(stmt_sequence());
13.       tmp->children.append(selfexpTree);
14.       match("endfor");
15.       return tmp;
16.   }
```

4.2.8 while-stmt 文法

函数首先调用 match 函数，匹配当前的"while"标记，并确认它与预期的"while"关键字匹配。然后，函数创建一个新的 SyntaxNode 对象，用"while"字符串初始化，这个节点将作为 while 循环的根节点。接着，函数匹配左括号"("，这表示 while 循环条件部分的开始。函数调用 exp 函数来解析循环条件表达式，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。函数匹配右括号")"，表示条件部分的结束。函数调用 stmt_sequence 函数来解析 while 循环体中的语句序列，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。最后，函数匹配"endwhile"标记，这表示 while 循环的结束。函数返回 tmp 指针，它指向表示整个 while 循环的 AST 的根节点。

```
1.    SyntaxTree MainWindow::while_stmt()
2.    {
3.        match("while");
4.        SyntaxTree tmp = new SyntaxNode("while");
5.        match("(");
6.        tmp->children.append(exp());
7.        match(")");
8.        tmp->children.append(stmt_sequence());
9.        match("endwhile");
10.       return tmp;
11.   }
```

4.2.9 assign-stmt 文法

首先，函数将为 TOKEN 的值存储在局部变量 tmpToken 中。接着，函数调用 match 函数，匹配一个标识符。函数调用 assign_sub_stmt 函数来解析赋值操作的其余部分（通常是表达式），并将返回的 AST 节点存储在 equalTree 指针中。如果 assign_sub_stmt 函数返回 nullptr（表示没有成功解析出表达式），则 assign_stmt 函数也返回 nullptr。如果表达式解析成功，函数创建一个新的 SyntaxNode 对象，包含标识符信息，并将其添加到 equalTree 的 children 数组的前端。这意味着赋值语句的 AST 中，标识符（作为赋值的目标）成为赋值操作的左子树，而表达式（作

为赋值的源）成为右子树。最后，函数返回 equalTree 指针，它指向表示整个赋值语句的 AST 的根节点。

```
1.    SyntaxTree MainWindow::assign_stmt()
2.    {
3.        QString tmpToken = TOKEN;
4.        match(3);    // 匹配标识符
5.        SyntaxTree equalTree = assign_sub_stmt();
6.        if (equalTree == nullptr) return nullptr;
7.        equalTree->children.push_front(new SyntaxNode
    ("id(" + tmpToken + ")"));    // 赋值左子树为标识符，
    右子树为表达式
8.        return equalTree;
9.    }
```

4.2.10　read-stmt 文法

函数首先调用 match 函数，与预期的"read"关键字匹配。然后，函数创建一个新的 SyntaxNode 对象，用"read"字符串初始化，这个节点将作为 read 语句的根节点。接着，函数将 TOKEN 的值存储在局部变量 tmpToken 中。函数调用 match 函数，并传入数字 3 作为参数。代表一个标识符的类型标记。函数创建一个新的 SyntaxNode 对象，包含标识符信息，并将其添加到 tmp 节点的 children 数组中。这意味着 read 语句的 AST 中，标识符（作为读取操作的目标）成为根节点的子节点。最后，函数返回 tmp 指针，它指向表示整个 read 语句的 AST 的根节点。

```
1.    SyntaxTree MainWindow::read_stmt()
2.    {
3.        match("read");
4.        SyntaxTree tmp = new SyntaxNode("read");
5.        QString tmpToken = TOKEN;
6.        match(3);    // 匹配标识符
7.        tmp->children.append(new SyntaxNode("id(" +
    tmpToken + ")"));
8.        return tmp;
9.    }
```

4.2.11　write-stmt 文法

函数首先调用 match 函数，与预期的"write"关键字匹配。然后，函数创建一个新的 SyntaxNode 对象，用"write"字符串初始

化，这个节点将作为write语句的根节点。函数调用write_sub_stmt函数来解析 write 语句中的子语句（一个或多个表达式，指定了要写入输出的内容），并将返回的 AST 节点添加到 tmp 节点的 children 数组中。最后，函数返回 tmp 指针，它指向表示整个 write 语句的 AST 的根节点。

```
SyntaxTree MainWindow::write_stmt()
{
    match("write");
    SyntaxTree tmp = new SyntaxNode("write");
    tmp->children.append(write_sub_stmt());
    return tmp;
}
```

4.2.12 write-sub-stmt 文法

函数首先检查一个名为 curType 的变量是否等于 1。表示字符类型。如果是字符类型，函数创建一个新的 SyntaxNode 对象，用字符类型的字符串表示初始化（例如，"char('A')"）。然后，函数调用 match 函数，并传入数字 1 作为参数。用来确认当前的标记与预期的字符类型匹配，并消耗这个标记。最后，返回新创建的 SyntaxNode 对象的指针。如果 curType 不等于 1，表明当前处理的不是一个字符类型。在这种情况下，函数调用 exp 函数来解析一个表达式，并返回解析得到的 AST 节点的指针。

```
SyntaxTree MainWindow::write_sub_stmt()
{
    if (curType == 1) { // 字符
        SyntaxTree tmp = new SyntaxNode("char("
  + TOKEN + ")");
        match(1);
        return tmp;
    } else {
        return exp();
    }
}
```

4.2.13 assign-sub-stmt 文法

函数首先检查名为 TOKEN 的变量是否等于":="。这个变量代表当前处理的语法标记，其中":="通常表示赋值操作。如果是

赋值操作，函数创建一个新的 SyntaxNode 对象，用赋值操作符的字符串表示初始化。然后，函数调用 match 函数，用来确认当前的标记与预期的赋值操作符":="匹配，并消耗这个标记。接着，函数调用 exp 函数来解析赋值操作的右侧表达式，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。最后，返回新创建的表示赋值操作的 SyntaxNode 对象的指针。如果 TOKEN 不是赋值操作符，函数接着检查 TOKEN 是否等于"==",这通常表示相等性比较。如果是相等性比较，函数创建一个新的 SyntaxNode 对象，用比较操作符的字符串表示初始化。然后，函数调用 match 函数，用来确认当前的标记与预期的比较操作符"=="匹配，并消耗这个标记。接着，函数调用 re 函数来解析相等性比较的右侧表达式，并将返回的 AST 节点添加到 tmp 节点的 children 数组中。注意，这里的 re 函数是一个错误，通常应该是 exp 函数来解析表达式。最后，返回新创建的表示相等性比较操作的 SyntaxNode 对象的指针。如果当前的 TOKEN 既不表示赋值操作也不表示相等性比较，函数将执行错误处理。ERROR(0); 和 ERROR(4);：调用错误处理函数，记录错误信息或执行其他错误处理逻辑。然后，函数返回 nullptr，表示没有成功解析出有效的赋值或比较操作。

```
1.    SyntaxTree MainWindow::assign_sub_stmt()
2.    {
3.        if (TOKEN == ":=") {
4.            SyntaxTree tmp = new SyntaxNode(":=");
5.            match(":=");
6.            tmp->children.append(exp());
7.            return tmp;
8.        } else if (TOKEN == "==") {
9.            SyntaxTree tmp = new SyntaxNode("==");
10.           match("==");
11.           tmp->children.append(re());
12.           return tmp;
13.       } else {
14.           ERROR(0);
15.           ERROR(4);
16.           return nullptr;
```

```
        }
    }
```

4.2.14 re 文法

函数首先调用 orre 函数来解析逻辑“或”表达式中的一个子表达式，并将返回的 AST 节点赋值给 tmp。然后，函数进入一个循环，只要当前的语法标记 TOKEN 是竖线"|"（通常用于表示逻辑“或”操作），循环就会继续。在循环体内，创建一个新的 SyntaxNode 对象，用逻辑“或”操作符的字符串表示初始化。调用 match 函数，用来确认当前的标记与预期的逻辑“或”操作符"|"匹配，并消耗这个标记。将之前解析的子表达式（存储在 tmp 中）添加到新节点 newTmp 的子节点列表中。再次调用 orre 函数来解析循环体内的下一个子表达式，并将返回的 AST 节点也添加到 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的复合表达式节点 newTmp，以便下一次循环时作为左子树使用。循环结束后，返回 tmp 指针，它指向表示整个逻辑“或”表达式的 AST 的根节点。

```
SyntaxTree MainWindow::re()
{
    SyntaxTree tmp = orre();
    while (TOKEN == "|") {
        SyntaxTree newTmp = new SyntaxNode("|");
        match("|");
        newTmp->children.append(tmp);
        newTmp->children.append(orre());
        tmp = newTmp;
    }
    return tmp;
}
```

4.2.15 orre 文法

函数首先调用 conre 函数来解析逻辑“与”表达式中的一个子表达式，并将返回的 AST 节点赋值给 tmp。然后，函数进入一个循环，只要当前的语法标记 TOKEN 是和号"&"（通常用于表示逻辑“与”操作），循环就会继续。在循环体内，创建一个新的 SyntaxNode 对象，用逻辑“与”操作符的字符串表示初始

化。调用 match 函数，用来确认当前的标记与预期的逻辑“与”操作符"&"匹配，并消耗这个标记。将之前解析的子表达式（存储在 tmp 中）添加到新节点 newTmp 的子节点列表中。再次调用 conre 函数来解析循环体内的下一个子表达式，并将返回的 AST 节点也添加到 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的复合表达式节点 newTmp，以便下一次循环时作为左子树使用。循环结束后，返回 tmp 指针，它指向表示整个逻辑“与”表达式的 AST 的根节点。

```
1.    SyntaxTree MainWindow::orre()
2.    {
3.        SyntaxTree tmp = conre();
4.        while (TOKEN == "&") {
5.            SyntaxTree newTmp = new SyntaxNode("&");
6.            match("&");
7.            newTmp->children.append(tmp);
8.            newTmp->children.append(conre());
9.            tmp = newTmp;
10.       }
11.       return tmp;
12.   }
```

4.2.16 conre 文法

函数首先调用 repre 函数来解析一个基本的正则表达式片段，并将返回的 AST 节点赋值给 tmp。接着，函数检查当前的语法标记 TOKEN 是否是井号"#"或问号"?"。这两个符号分别表示闭包（如#）和可选（如?）操作。如果是闭包或可选操作之一，函数创建一个新的 SyntaxNode 对象，用当前的 TOKEN 字符串表示初始化。调用 match 函数，用来确认当前的标记与预期的操作符匹配，并消耗这个标记。这确保了语法分析的准确性。将之前解析的基本表达式片段（存储在 tmp 中）作为子节点添加到新节点 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的闭包或可选操作节点 newTmp，这样在函数返回时，它将指向包含闭包或可选操作的完整表达式。最后，函数返回 tmp 指针，它指向表示整个闭包或可选操作表达式的 AST 的根节点。

```
1.    SyntaxTree MainWindow::conre()
2.    {
3.        SyntaxTree tmp = repre();
4.        if (TOKEN == "#" || TOKEN == "?") {
5.            SyntaxTree newTmp = new SyntaxNode(TOKEN)
    ;
6.            match(TOKEN);       // 捕获闭包符号或可选符
    号
7.            newTmp->children.append(tmp);
8.            tmp = newTmp;
9.        }
10.       return tmp;
11.   }
```

4.2.17　repre 文法

函数首先检查当前的语法标记 TOKEN 是否是左括号"("。如果是，函数调用 match 函数来消耗这个左括号标记。然后调用 re 函数来递归地解析括号内的正则表达式，并返回得到的 AST 节点，存储在 tmp 中。之后，函数再次调用 match 函数来消耗对应的右括号")"标记。最后，返回 tmp 指针，它指向括号内表达式的 AST。如果 TOKEN 不是左括号，函数接着检查一个名为 curType 的变量是否等于 1，这表示当前标记是一个字符类型。如果是，函数创建一个新的 SyntaxNode 对象，用字符类型的字符串表示初始化，其中包含当前的 TOKEN 值。调用 match 函数，并传入数字 1 作为参数，用来确认当前的标记与预期的字符类型匹配，并消耗这个标记。返回新创建的表示字符的 SyntaxNode 对象的指针。如果 curType 不等于 1，函数再检查 curType 是否等于 3，这表示当前标记是一个标识符类型。如果是，函数创建一个新的 SyntaxNode 对象，用标识符类型的字符串表示初始化，其中包含当前的 TOKEN 值。调用 match 函数，并传入数字 3 作为参数，是用来确认当前的标记与预期的标识符类型匹配，并消耗这个标记。返回新创建的表示标识符的 SyntaxNode 对象的指针。如果以上条件都不满足，函数将执行错误处理。调用 ERROR 函数两次，记录错误信息或执行其他错误处理逻辑。返回 nullptr，表示没有成功解析出有效的正则表达式基本片段。

```
1.    SyntaxTree MainWindow::repre()
2.    {
3.        if (TOKEN == "(") {
4.            match("(");
5.            SyntaxTree tmp = re();
6.            match(")");
7.            return tmp;
8.        } else if (curType == 1) {      // 字符
9.            SyntaxTree tmp = new SyntaxNode("char("
   + TOKEN + ")");
10.           match(1);
11.           return tmp;
12.       } else if (curType == 3) {      // 标识符
13.           SyntaxTree tmp = new SyntaxNode("id(" +
   TOKEN + ")");
14.           match(3);
15.           return tmp;
16.       } else {
17.           ERROR(0);
18.           ERROR(1);
19.           return nullptr;
20.       }
21.   }
```

4.2.18 exp 文法

函数首先调用 simple_exp 函数来解析一个简单表达式（如一个项或由算术运算符连接的因子），并将返回的 AST 节点赋值给 tmp。接着，函数检查当前的语法标记 TOKEN 是否是一个比较操作符。如果是，函数执行以下步骤：创建一个新的 SyntaxNode 对象，用当前的比较操作符的字符串表示初始化。调用 match 函数，用来确认当前的标记与预期的比较操作符匹配，并消耗这个标记。将之前解析的简单表达式（存储在 tmp 中）作为左子树添加到新节点 newTmp 的子节点列表中。再次调用 simple_exp 函数来解析比较操作符之后的简单表达式，并将返回的 AST 节点作为右子树添加到 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的比较表达式节点 newTmp，这样在函数返回时，它将指向包含比较操作的完整表达式。最后，函数返回 tmp 指针，它指向表示整个比较表达式的 AST 的根节点。

```
SyntaxTree MainWindow::exp()
{
    SyntaxTree tmp = simple_exp();
    if (TOKEN == "<" || TOKEN == ">" || TOKEN == "=" || TOKEN == ">=" || TOKEN == "<=" || TOKEN == "<>") {
        SyntaxTree newTmp = new SyntaxNode(TOKEN);
        match(TOKEN);
        newTmp->children.append(tmp);
        newTmp->children.append(simple_exp());
        tmp = newTmp;
    }
    return tmp;
}
```

4.2.19 simple-exp 文法

函数首先调用 term 函数来解析一个项（是由因子通过乘法或除法操作符连接的序列），并将返回的 AST 节点赋值给 tmp。接着，函数进入一个循环，只要当前的语法标记 TOKEN 是加号"+"或减号"-"，循环就会继续。在循环体内，创建一个新的 SyntaxNode 对象，用当前的加法或减法操作符的字符串表示初始化。调用 match 函数，用来确认当前的标记与预期的加法或减法操作符匹配，并消耗这个标记。将之前解析的项（存储在 tmp 中）作为左子树添加到新节点 newTmp 的子节点列表中。再次调用 term 函数来解析操作符之后的项，并将返回的 AST 节点作为右子树添加到 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的加法或减法表达式节点 newTmp，以便下一次循环时作为左子树使用。循环结束后，返回 tmp 指针，它指向表示整个简单表达式的 AST 的根节点。

```
SyntaxTree MainWindow::simple_exp()
{
    SyntaxTree tmp = term();
    while (TOKEN == "+" || TOKEN == "-") {
        SyntaxTree newTmp = new SyntaxNode(TOKEN);
        match(TOKEN);
        newTmp->children.append(tmp);
        newTmp->children.append(term());
```

```
            tmp = newTmp;
        }
        return tmp;
    }
```

4.2.20 term 文法

函数首先调用 factor 函数来解析一个因子（一个数值、变量、括号内的表达式等），并将返回的 AST 节点赋值给 tmp。接着，函数进入一个循环，只要当前的语法标记 TOKEN 是乘号"*"、除号"/"或百分号"%"（通常用于表示取模操作），循环就会继续。在循环体内，创建一个新的 SyntaxNode 对象，用当前的操作符的字符串表示初始化。调用 match 函数，用来确认当前的标记与预期的操作符匹配，并消耗这个标记。将之前解析的因子（存储在 tmp 中）作为左子树添加到新节点 newTmp 的子节点列表中。再次调用 factor 函数来解析操作符之后的因子，并将返回的 AST 节点作为右子树添加到 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的项节点 newTmp，以便下一次循环时作为左子树使用。循环结束后，返回 tmp 指针，它指向表示整个项的 AST 的根节点。

```
SyntaxTree MainWindow::term()
{
    SyntaxTree tmp = factor();
    while (TOKEN == "*" || TOKEN == "/" || TOKEN == "%") {
        SyntaxTree newTmp = new SyntaxNode(TOKEN);
        match(TOKEN);
        newTmp->children.append(tmp);
        newTmp->children.append(factor());
        tmp = newTmp;
    }
    return tmp;
}
```

4.2.21 factor 文法

函数首先调用 resexp 函数来解析一个基本表达式（如数值、变量或括号内的表达式等），并将返回的 AST 节点赋值给 tmp。

接着，函数进入一个循环，只要当前的语法标记 TOKEN 是指数（幂）操作符"^"，循环就会继续。在循环体内，创建一个新的 SyntaxNode 对象，用当前的指数操作符的字符串表示初始化。调用 match 函数，用来确认当前的标记与预期的指数操作符"^"匹配，并消耗这个标记。将之前解析的基本表达式（存储在 tmp 中）作为基数添加到新节点 newTmp 的子节点列表中。再次调用 resexp 函数来解析指数操作符之后的表达式，并将返回的 AST 节点作为指数添加到 newTmp 的子节点列表中。更新 tmp 指针，使其指向新的因子节点 newTmp，以便下一次循环时作为基数使用。循环结束后，返回 tmp 指针，它指向表示整个因子的 AST 的根节点。

```
1.    SyntaxTree MainWindow::factor()
2.    {
3.        SyntaxTree tmp = resexp();
4.        while (TOKEN == "^") {
5.            SyntaxTree newTmp = new SyntaxNode(TOKEN)
    ;
6.            match(TOKEN);
7.            newTmp->children.append(tmp);
8.            newTmp->children.append(resexp());
9.            tmp = newTmp;
10.       }
11.       return tmp;
12.   }
```

4.2.22　selfexp 文法

函数首先检查当前的语法标记 TOKEN 是否是自增"++"或自减"--"操作符。如果是，将操作符存储在临时变量 tmpToken 中。调用 match 函数，用来确认当前的标记与预期的操作符匹配，并消耗这个标记。接着，检查名为 curType 的变量是否等于 3，这表示当前标记是一个标识符类型。如果是标识符，函数执行以下步骤：创建一个新的 SyntaxNode 对象，用当前的操作符（"++"或"--"）的字符串表示初始化。为 tmp 节点添加一个子节点，这个子节点代表紧随操作符后的标识符。调用 match 函数，并传入数字 3 作为参数，是用来确认当前的标记与预期的标识符类型匹

配，并消耗这个标记。返回新创建的表示自增或自减操作的 SyntaxNode 对象的指针。如果 curType 不等于 3，即当前标记不是一个标识符。调用错误处理函数，记录错误信息或执行其他错误处理逻辑。返回 nullptr，表示没有成功解析出自增或自减操作。如果最初的 TOKEN 检查失败，即当前的语法标记不是"++"或"--"。调用错误处理函数，记录错误信息或执行其他错误处理逻辑。返回 nullptr，表示没有成功解析出自增或自减操作。

```
1.    SyntaxTree MainWindow::selfexp()
2.    {
3.        if (TOKEN == "++" || TOKEN == "--") {
4.            QString tmpToken = TOKEN;
5.            match(TOKEN);
6.            if (curType == 3) { // 标识符
7.                SyntaxTree tmp = new SyntaxNode(tmpT
   oken);
8.                tmp->children.append(new SyntaxNode(
   "id(" + TOKEN + ")"));
9.                match(3);
10.               return tmp;
11.           } else {
12.               ERROR(1);
13.               return nullptr;
14.           }
15.       } else {
16.           ERROR(0);
17.           return nullptr;
18.       }
19.   }
```

4.2.23 resexp 文法

函数首先检查当前的语法标记 TOKEN 是否是左括号"("。如果是，函数执行以下步骤：调用 match 函数来消耗左括号标记。调用 exp 函数来解析括号内的表达式，并返回得到的 AST 节点，存储在 tmp 中。调用 match 函数来消耗右括号标记。返回 tmp 指针，它指向括号内表达式的 AST。如果 TOKEN 不是左括号，函数接着检查名为 curType 的变量是否等于 2，这可能表示当前标记是一个数字类型。如果是数字，函数执行以下步骤：创建一

个新的 SyntaxNode 对象，用数字类型的字符串表示初始化，其中包含当前的 TOKEN 值。调用 match 函数，并传入数字 2 作为参数，可能是用来确认当前的标记与预期的数字类型匹配，并消耗这个标记。返回新创建的表示数字的 SyntaxNode 对象的指针。如果 curType 不等于 2，函数再检查 curType 是否等于 3，这可能表示当前标记是一个标识符类型。如果是标识符，函数执行以下步骤：创建一个新的 SyntaxNode 对象，用标识符类型的字符串表示初始化，其中包含当前的 TOKEN 值。调用 match 函数，并传入数字 3 作为参数，可能是用来确认当前的标记与预期的标识符类型匹配，并消耗这个标记。返回新创建的表示标识符的 SyntaxNode 对象的指针。如果以上条件都不满足，但当前的 TOKEN 是自增"++"或自减"--"操作符。调用 selfexp 函数来解析自增或自减操作，并返回得到的 AST 节点。如果所有条件都不满足，函数将执行错误处理。调用错误处理函数，可能记录错误信息或执行其他错误处理逻辑。返回 nullptr，表示没有成功解析出有效的基元表达式。

```
1.    SyntaxTree MainWindow::resexp()
2.    {
3.        if (TOKEN == "(") {
4.            match("(");
5.            SyntaxTree tmp = exp();
6.            match(")");
7.            return tmp;
8.        } else if (curType == 2) {      // 数字
9.            SyntaxTree newTmp = new SyntaxNode("numbe
   r(" + TOKEN + ")");
10.           match(2);
11.           return newTmp;
12.       } else if (curType == 3) {      // 标识符
13.           SyntaxTree newTmp = new SyntaxNode("id("
   + TOKEN + ")");
14.           match(3);
15.           return newTmp;
16.       } else if (TOKEN == "++" || TOKEN == "--") {
17.           return selfexp();
18.       } else {
19.           ERROR(0);
20.           ERROR(1);
21.           return nullptr;
22.       }
23.   }
```

5 测试及结果展示

5.1 测试 1：修改语法中的 if

测试 1 代码：

```
{ Sample program
in TINY language -
computes factorial
}
read x; { input an integer }
if (0<x) { don't compute if x <= 0 }
fact := 1;
repeat
fact := fact * x;
fact := fact + 1
until x = 10;
write fact { output factorial of x }
endif
```

测试 1 结果：

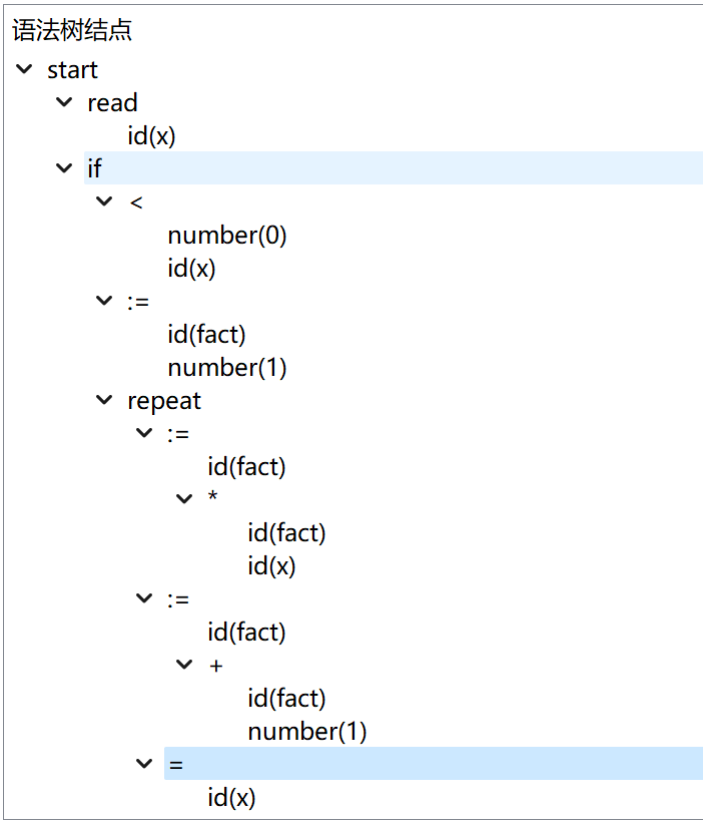

图 5 测试 1 部分结果展示

5.2 测试 2：for 语句测试

测试 2 代码：

```
{ Sample program
   in TINY language -
   computes factorial
}
read x; { input an integer }
if   (0<x)   { don't compute if x <= 0 }
   for( fact := 1; x>0;--x)
         fact := fact * x
   endfor;
   write fact   { output factorial of x }
endif
```

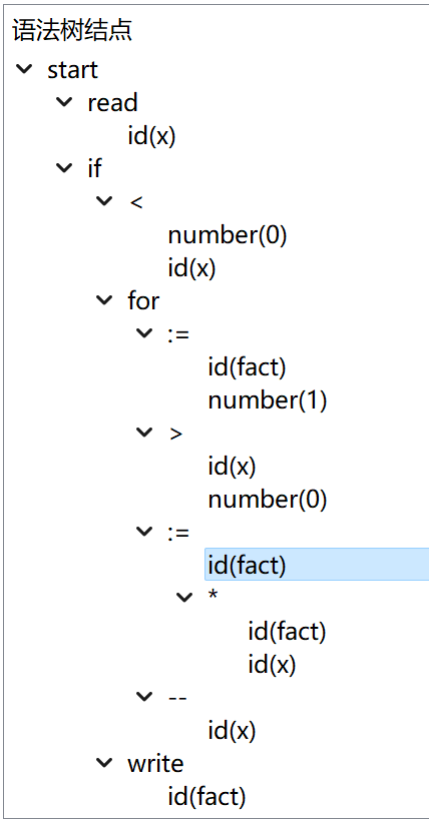

图 6 测试 2 部分结果展示

5.3 测试 3：while 语句测试

测试 3 代码：

```
{ Sample program
   in TINY language -
   computes factorial
}
read x; { input an integer }
if   (0<x)   { don't compute if x <= 0 }
   fact := 1;
   while (fact < 10)
       x := x * fact;
       fact := fact + 1
   endwhile;
   write fact   { output factorial of x }
endif
```

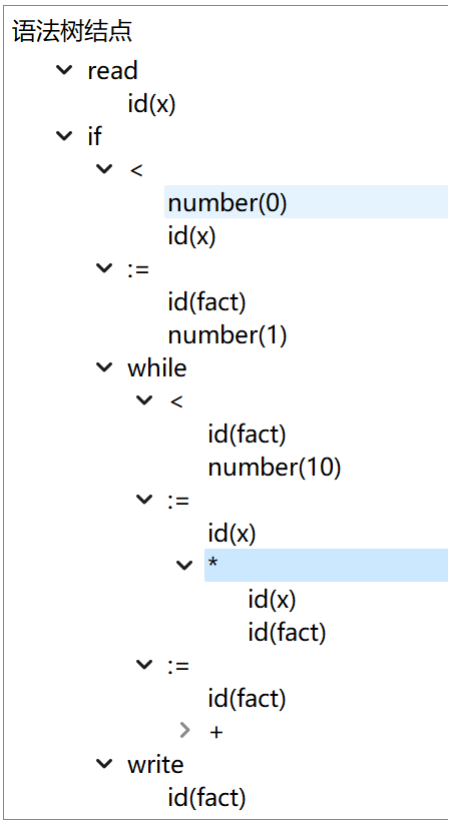

图 7 测试 3 部分结果展示

5.4 测试 4：正则表达式赋值

```
re == ('a'|'b'&'c')#&('d'&'e'|'f')?&'g'
```

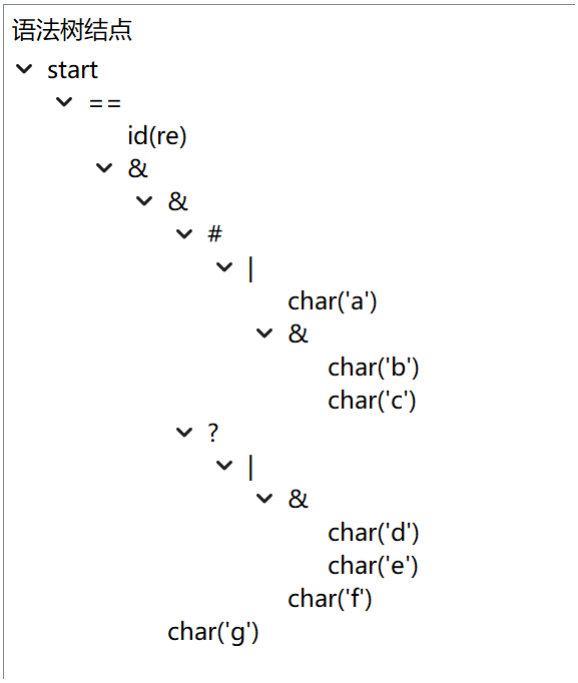

图 8 测试 4 部分结果展示

5.5 测试 5：扩充后的算术表达式

```
x := 114514^233;
y := ++x%(1e9+7)^x
```

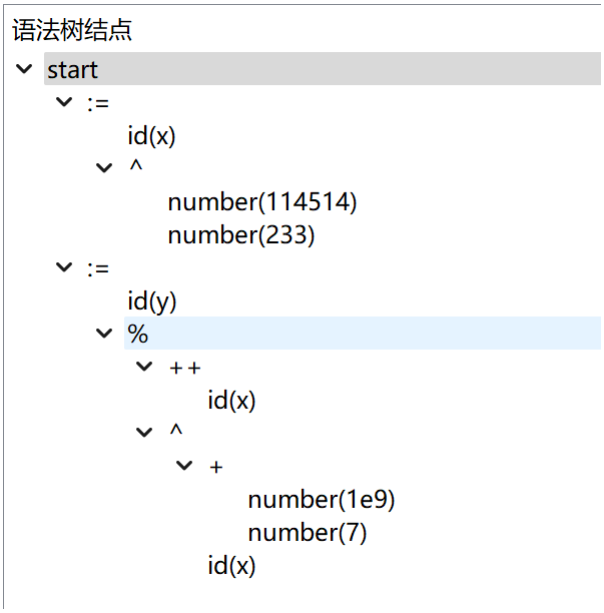

图 9 测试 5 部分结果展示

5.6 测试 6：扩充判断语句

```
com1 := 1 < 2;
com2 := 2 > 1;
if (com1 <> com2)
com3 := com1;
com4 := com2
endif;
if (com3 = com4)
com5 := com4
endif;
write com4 >= 0;
write com5 <= 1
```

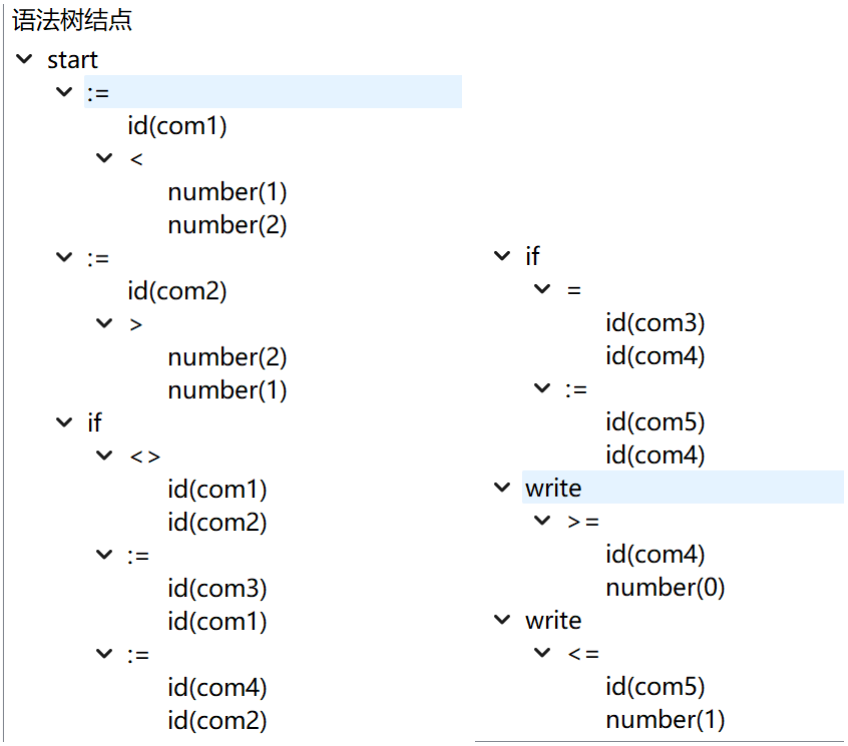

图 10 测试 6 部分结果展示

四、实验总结（心得体会）

1. 理论与实践相结合的重要性：通过实验，我更加深刻地理解了编译原理中的词法分析和语法分析，将理论知识应用到实际问题中。

2. 细节的重要性：在实现词法分析器和语法分析器的过程中，我意识到了对语言细节的精确把握对于正确实现分析器的重要性。

3. 调试的挑战与乐趣：在遇到错误和异常时，我学会了如何使用调试工具来定位问题，并享受解决问题后的成就感。

4. 语言设计的灵活性：通过为 Tiny 语言增加新的语法规则，我体会到了语言设计的灵活性和创造性，以及对现有语言进行改进的挑战。

5. 持续学习的必要性：实验让我认识到了编译原理领域的广博和深奥，激发了我继续深入学习的兴趣。

6. 对编程语言的深入理解：通过扩充 Tiny 语言，我对编程语言的内部机制和设计哲学有了更深入的认识。

五、参考文献：

[1] 编译原理：TINY 语言的语法、词法单元与文法的最全总结-CSDN 博客

[2] 编译原理（3）——for 循环_编译原理实验三语法分析 for 循环-CSDN 博客

[3] AST(抽象语法树)超详细-CSDN 博客

六、附录一：扩充 Tiny 语言分词代码实现

```
1.    QVector<QPair<QString, LexerInfo> > lexer(QString content
   )
2.    {
3.
4.        // 获取关键字与运算符
5.        QFile file(":/mapping.json");
6.        QHash<QString, QString> hash;
7.        if (file.open(QIODevice::ReadOnly | QIODevice::Text))
    {
8.            QByteArray data = file.readAll();
9.            QJsonDocument jsonDocument = QJsonDocument::fromJ
   son(data);
10.
11.           if (!jsonDocument.isNull()) {
12.               QJsonObject jsonObject = jsonDocument.object(
   );
13.
14.               for (auto it = jsonObject.begin(); it != json
   Object.end(); ++it) {
15.                   QString key = it.key();
16.                   QString value = it.value().toString();
17.                   hash.insert(key, value);
18.               }
19.           }
20.       }
21.
22.       // 开始分词
23.       QVector<QPair<QString, LexerInfo>> tokenList;
24.       QStringList lines = content.split("\n");
25.       bool finishCom = false;
26.       int cachePos = 0;
27.       for (int index = 0; index < lines.size(); index++) {
28.           const QString& line = lines[index];
29.           int pos = finishCom ? cachePos : 0;
30.           int len = line.size();
31.
32.           finishCom = false;
```

```
33.
34.             // 跳过空格与tab
35.             while (pos < len && (line[pos] == ' ' || line[pos
    ] == '\t')) pos++;
36.             if (pos == len) continue;
37.
38.             // 遍历单行
39.             while (pos < len) {
40.   //             qDebug() << line[pos] << ' ' << hash.contai
    ns(QString(line[pos])) << endl;
41.                 if (hash.contains(line[pos]) && hash[line[pos
    ]] == "separator") {
42.                     // 判断 (、)、; 等分隔符号
43.                     tokenList.append(qMakePair(QString(line[p
    os]), LexerInfo(0, index, pos)));
44.                     pos++;
45.                 } else if (pos != len && line[pos] == '\'') {
46.                     // 字符类型
47.                     int endIndex = line.indexOf(line[pos], po
    s + 1);
48.                     if (endIndex != -1) {
49.                         while (endIndex != -1 && line[endInde
    x - 1] == '\\') endIndex = line.indexOf(line[pos], endIndex
    + 1);
50.                         if (endIndex == -1) {
51.                             break;
52.                         }
53.                         QString str = line.mid(pos, endIndex
    - pos + 1);
54.                         tokenList.append(qMakePair(str, Lexer
    Info(1, index, pos)));
55.                         pos = endIndex + 1;
56.                     } else {
57.                         // 出现错误
58.                         break;
59.                     }
60.                 } else if (line[pos] == '+' || line[pos] == '
    -' || line[pos].isDigit()) {
61.                     // 数字类型或运算符
62.                     if (line[pos] == '+' || line[pos] == '-')
     {
63.                         // 判断 + 与 - 是运算符还是数字的一部分
64.
65.                         if (tokenList.back().second.lexerType
```

```
    != 5) {
                            // 在tiny语言中，上一个token不为运算符，则当前+-必定为运算符
                            tokenList.append(qMakePair(QString(line[pos]), LexerInfo(5, index, pos)));
                            pos++;
                            continue;
                        }
    //                    qDebug() << tokenList.back().first << ' ' << line[pos];
                        if (tokenList.back().first == line[pos] && tokenList.size() > 1 && tokenList[tokenList.size() - 2].second.lexerType != 2) {
    //                        qDebug() << pos << endl;
                            tokenList.pop_back();
                            tokenList.append(qMakePair(QString(line[pos] == '+' ? "++" : "--"), LexerInfo(5, index, pos - 1)));
                            pos++;
                            continue;
                        }
                        if (pos == len - 1 || !line[pos + 1].isDigit()) {
                            // + 与 - 位于最后一个位置或后续字符不为数字
                            tokenList.append(qMakePair(QString(line[pos]), LexerInfo(5, index, pos)));
                            pos++;
                            continue;
                        }
                    }

                    // 完整数字匹配
                    QString number = ((line[pos] == '+' || line[pos] == '-' ) ? line[pos] : QString(""));
                    if (line[pos] == '+' || line[pos] == '-') pos++;
                    while (pos < len && line[pos].isDigit()) number += line[pos++];
                    if (pos < len - 1 && line[pos] == '.' && line[pos + 1].isDigit()) number += line[pos++];
                    while (pos < len && line[pos].isDigit()) number += line[pos++];
                    if (pos < len - 1 && (line[pos] == 'E' ||
```

```
    line[pos] == 'e') && line[pos + 1].isDigit()) number += line[pos++];
                    while (pos < len && line[pos].isDigit()) number += line[pos++];
                    tokenList.append(qMakePair(QString(number), LexerInfo(2, index, pos - number.size())));

                } else if (line[pos].isLetter() || line[pos] == '_') {
                    // 标识符或关键字
                    QString identifier;
                    while (pos < len && (line[pos].isDigit() || line[pos].isLetter() || line[pos] == '_')) identifier += line[pos++];

                    // 判断是否为关键词
                    if (hash.contains(identifier) && hash[identifier] == "keyword") {
                        // 关键词
                        tokenList.append(qMakePair(identifier, LexerInfo(4, index, pos - identifier.size())));
                    } else {
                        // 标识符
                        tokenList.append(qMakePair(identifier, LexerInfo(3, index, pos - identifier.size())));
                    }

                } else if (line[pos] == '{') {
                    // 注释
                    int indexCnt = index, posCnt = pos;
                    while (indexCnt < lines.size()) {

                        // 查找注释结束位置
                        int endCom = lines[indexCnt].indexOf('}', pos);
                        if (endCom != -1) {
                            tokenList.append(qMakePair(lines[indexCnt].mid(posCnt, endCom - posCnt + 1), LexerInfo(6, indexCnt, posCnt)));
                            posCnt = endCom + 1;
                            break;
                        }

                        // 没找到就找下一行
```

```
                    tokenList.append(qMakePair(lines[indexCnt].right(lines[indexCnt].size() - posCnt), LexerInfo(6, indexCnt, posCnt)));
                    posCnt = 0;
                    indexCnt++;

                }
                index = indexCnt - 1;
                cachePos = posCnt;
                finishCom = true;
                break;

            } else {
                // 运算符
                if (pos < len - 1 && hash.contains(line.mid(pos, 2)) && hash[line.mid(pos, 2)] == "operator") {
                    // 双字符运算符
                    tokenList.append(qMakePair(line.mid(pos, 2), LexerInfo(5, index, pos)));
                    pos += 2;
                } else if (hash.contains(line[pos]) && hash[line[pos]] == "operator") {
                    // 单字符运算符
                    tokenList.append(qMakePair(line[pos], LexerInfo(5, index, pos)));
                    pos++;
                } else {
                    // 其他
                    pos++;
                }
            }

        }
    }

//    for (const auto &token : tokenList) {
//        qDebug() << token.first << ' ' << token.second.lexerType;
//    }

    return tokenList;
}
```

七、附录二：主页面代码

```
MainWindow::MainWindow(QWidget *parent)
```

```
2.        : QMainWindow(parent)
3.        , TOKEN(""), curType(6), curRow(0), curCol(0), tokenI
   ndex(0), root(nullptr)
4.        , ui(new Ui::MainWindow)
5.    {
6.        ui->setupUi(this);
7.
8.        // 基本布局信息
9.        this->setWindowTitle("编译原理实验三：扩展 Tiny 语言的语法
   树生成");
10.       QScreen *screen = QGuiApplication::primaryScreen();
11.       QRect screenGeometry = screen->geometry();
12.       int screenWidth = screenGeometry.width();
13.       int screenHeight = screenGeometry.height();
14.       int newWidth = screenWidth * 0.64;
15.       int newHeight = screenHeight * 0.48;
16.       this->resize(newWidth, newHeight);
17.
18.       // 设置表格列
19.       ui->wordWidget->setColumnCount(2);
20.       ui->wordWidget->setHorizontalHeaderLabels(QStringList
   () << "token" << "类型");
21.       ui->wordWidget->horizontalHeader()->setStretchLastSec
   tion(true); // 最后一列自适应宽度
22.       ui->errorWidget->setColumnCount(3);
23.       ui->errorWidget->setHorizontalHeaderLabels(QStringLis
   t() << "错误说明" << "行" << "列");
24.       ui->errorWidget->horizontalHeader()->setStretchLastSe
   ction(true); // 最后一列自适应宽度
25.       ui->treeWidget->setColumnCount(1);
26.
27.       // 上传文件
28.       connect(ui->uploadButton, &QPushButton::clicked, this
   , [&]() {
29.           QString fileName = QFileDialog::getOpenFileName(t
   his, "选择 Tiny 源程序");
30.           if (!fileName.isEmpty()) {
31.               QFile file(fileName);
32.               if (file.open(QIODevice::ReadOnly | QIODevice
   ::Text)) {
33.                   QTextStream in(&file);
34.                   in.setCodec("UTF-8");
35.                   ui->textEdit->setText(in.readAll());
36.                   file.close();
```

```
37.                 }
38.             }
39.         });
40.
41.         // 词法分析与语法分析
42.         connect(ui->syntaxButton, &QPushButton::clicked, this
    , [&]() {
43.
44.             // 重置程序
45.             QString content = ui->textEdit->toPlainText();
46.             reset();
47.             ui->textEdit->setText(content);
48.
49.             // 分词
50.             tokenList = lexer(content);
51.
52.             // 分词展示
53.             ui->wordWidget->setRowCount(tokenList.size());
54.             ui->wordWidget->setEditTriggers(QAbstractItemView
    ::NoEditTriggers);
55.             for (int i = 0; i < tokenList.size(); i++) {
56.                 ui->wordWidget->setItem(i, 0, new QTableWidge
    tItem(tokenList[i].first));
57.                 ui->wordWidget->setItem(i, 1, new QTableWidge
    tItem(getTypeString(tokenList[i].second.lexerType)));
58.             }
59.             for (int row = 0; row < ui->wordWidget->rowCount(
    ); ++row) {
60.                 QTableWidgetItem *item = ui->wordWidget->item
    (row, 1);
61.                 if (item) item->setTextAlignment(Qt::AlignCen
    ter);
62.             }
63.             ui->wordWidget->show();
64.
65.             // 去除所有注释 token
66.             QVector<QPair<QString, LexerInfo>> tmpList = toke
    nList;
67.             tokenList.clear();
68.             for (const auto &pair: tmpList) {
69.                 if (pair.second.lexerType != 6) {
70.                     tokenList.append(pair);
71.                 }
72.             }
```

```
73.
74.          if (!tokenList.isEmpty()){
75.
76.              // 语法树构造
77.              getToken();
78.              root = program();
79.              if (tokenIndex != tokenList.size() + 1) {
80.                  ERROR(0);
81.              }
82.
83.              // 错误展示
84.              ui->errorWidget->setRowCount(errorList.size()
   );
85.              for (int i = 0; i < errorList.size(); i++) {
86.                  ui->errorWidget->setItem(i, 0, new QTable
   WidgetItem(errorList[i].info));
87.                  ui->errorWidget->setItem(i, 1, new QTable
   WidgetItem(QString::number(errorList[i].row)));
88.                  ui->errorWidget->setItem(i, 2, new QTable
   WidgetItem(QString::number(errorList[i].column)));
89.              }
90.              for (int row = 0; row < ui->errorWidget->rowC
   ount(); ++row) {
91.                  QTableWidgetItem *item = ui->errorWidget-
   >item(row, 0);
92.                  if (item) item->setTextAlignment(Qt::Alig
   nCenter);
93.                  item = ui->errorWidget->item(row, 1);
94.                  if (item) item->setTextAlignment(Qt::Alig
   nCenter);
95.                  item = ui->errorWidget->item(row, 2);
96.                  if (item) item->setTextAlignment(Qt::Alig
   nCenter);
97.              }
98.
99.              // 语法树展示
100.             if (errorList.empty()) {
101.                 ui->treeWidget->setHeaderLabels(QStringLi
   st()<<"语法树结点");
102.                 QTreeWidgetItem* topItem = new QTreeWidge
   tItem(QStringList() << "start");
103.                 ui->treeWidget->addTopLevelItem(topItem);
104.                 if (!root->children.isEmpty()) {
105.                     showTree(topItem, root->children[0]);
```

```
106.                 }
107.             } else {
108.                 ui->treeWidget->setHeaderLabels(QStringLi
    st()<<"语法有误");
109.             }
110.
111.         }
112.     });
113.
114.     // 重置程序
115.     connect(ui->resetButton, &QPushButton::clicked, this,
     [&]() {
116.         reset();
117.     });
118.
119.     // 保存源文件
120.     connect(ui->saveButton, &QPushButton::clicked, this,
    [=]() {
121.         QString fileName = QString("tiny") + QString::num
    ber(QDateTime::currentDateTime().toMSecsSinceEpoch()) + QStr
    ing(".txt");
122.         QFile file(fileName);
123.         if (file.open(QIODevice::WriteOnly | QIODevice::T
    ext)) {
124.             QTextStream out(&file);
125.             out << ui->textEdit->toPlainText();
126.             file.close();
127.             QMessageBox::information(this, "提示", "源文件
    保存为：" + fileName + "成功！", QMessageBox::Yes);
128.         } else {
129.             QMessageBox::warning(this, "提示", "源文件保存
    失败！", QMessageBox::Yes);
130.         }
131.     });
132. }
```