一、项目目的

任务一

设计一个词法分析程序，可以实现对高级程序设计语言的单词的正则表达式进行分析，生成相应的 NFA、DFA、最小化 DFA 和词法分析程序，并且能根据该程序语言的一个源程序，输出相应的单词编码。

任务二

设计一个语法分析程序，可以实现对高级程序设计语言的 BNF 文法进行分析，生成相应的 First 集合、Follow 集合、LR(0)的 DFA 和 SLR(1)的分析表，并且能根据该程序语言的任务一输出的单词编码，进行语句分析并生成语法树。

二、项目内容

任务一

（1）以文本文件的方式输入某一高级程序设计语言的所有单词对应的正则表达式，系统需要提供一个操作界面，让用户打开某一语言的所有单词对应正则表达式文本文件，该文本文件的具体格式可根据自己实际的需要进行定义。

（2）需要提供窗口以便用户可以查看转换得到的 NFA（可用状态转换表呈现）。

（3）需要提供窗口以便用户可以查看转换得到的 DFA（可用状态转换表呈现）

（4）需要提供窗口以便用户可以查看转换得到的最小化 DFA（可用状态转换表呈现）。

（5）需要提供窗口以便用户可以查看转换得到的词法分析程序（该分析程序需要用 C 语言描述）

（6）对要求（5）得到的源程序进行编译生成一个可执行程序，并以该高级程序设计语言的一个源程序进行测试，输出该源程序的单词编码。需要提供窗口以便用户可以进行查看该单词编码。

（7）对系统进行测试：（A）先以 TINY 语言的所有单词的正则表达式作为

文本来测试，生成一个 TINY 语言的词法分析程序；（B）接着对这个词法分析源程序利用 C/C++编译器进行编译，并生成可执行程序；（C）以 sample.tny 来测试，输出该 TINY 语言源程序的单词编码文件 sample.lex。

（8）要求应用程序为 windows 界面。

（9）书写完善的软件文档。

（10）以 mini-c 的词法进行测试，并以至少一个 mini-c 源程序进行词法分析的测试（该 mini-c 源程序需要自己根据 mini-c 词法和语法编写出来，类似与 sample.tny）。

任务二

（1）以文本文件的方式输入某一高级程序设计语言的所有语法对应的 BNF 文法，因此系统需要提供一个操作界面，让用户打开某一语言的所有语法对应的 BNF 文法的文本文件，该文本文件的具体格式可根据自己实际的需要进行定义。

（2）求出文法各非终结符号的 First 集合与 Follow 集合，并提供窗口以便用户可以查看这些集合结果。（可用两张表格的形式分别进行呈现）

（3）需要提供窗口以便用户可以查看文法对应的 LR(0)DFA 图。（可以用画图的方式呈现，也可用表格方式呈现该图点与边数据）。

（4）构造出 SLR(1)分析表，并需要提供窗口以便用户可以查看该结果（可用表格形式进行呈现）。

（5）采用 SLR(1)语法分析方法进行语法分析并生成相应的语法树，每个语句的语法树结构可根据实际的需要进行定义（语法树需要采用树状形式进行呈现）。

（6）以 TINY 语言的所有语法以及第一项任务的测试结 sample.lex 作为测试，并生成对应的语法树并呈现出来。

（7）要求应用程序应为 Windows 界面。

（8）书写完善的软件文档。

（9）选做内容：可以生成某种中间代码（具体的中间代码形式可以自定）。

（10）以 mini-c 的语法进行测试，并以项目一的源程序所生成的单词编码文件进行语法分析，生成对应的语法树。

三、项目文档

任务一

一、总体设计

1.1 需求概述

本项目要求设计一个分析程序，针对某一高级程序设计语言的单词的正则表达式，可以进行分析并生成相应的 NFA、DFA、最小化 DFA 和词法分析程序，并且能根据该程序语言的一个源程序，输出相应的单词编码。

1.2 处理流程

本项目的处理流程可以用以下流程图表示：

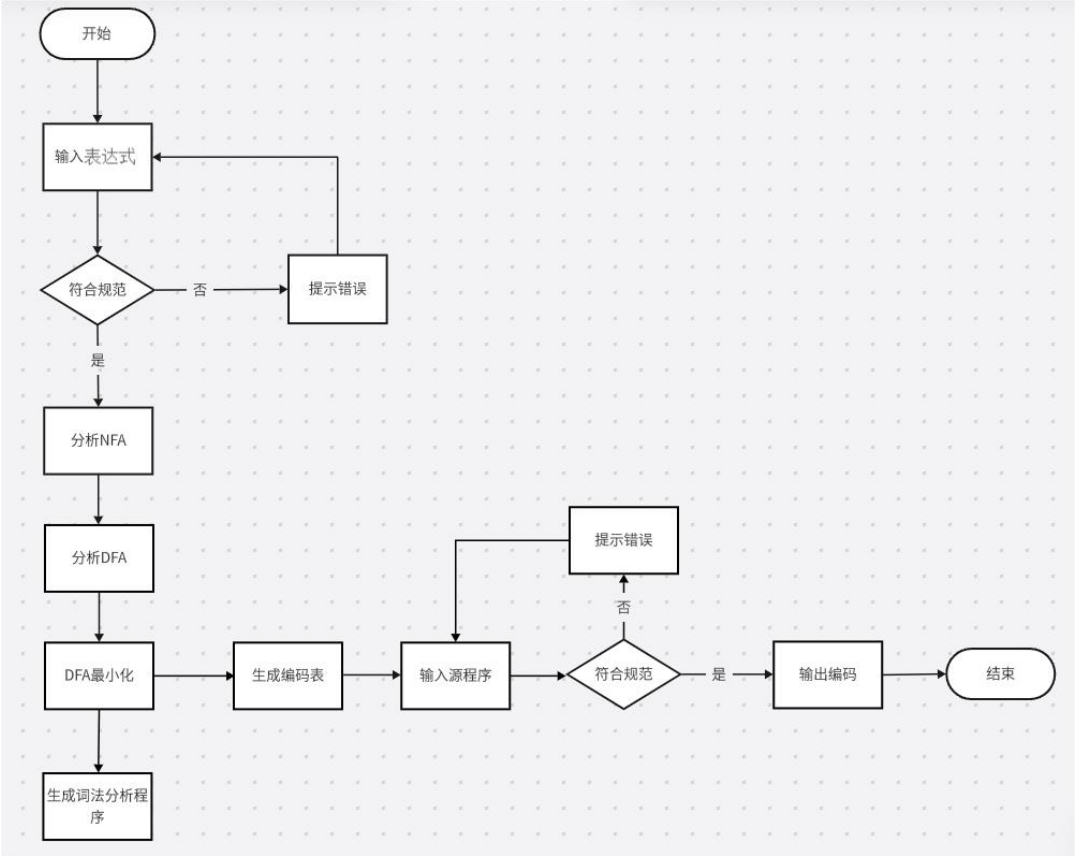

图 1　处理流程图

1.3 总体结构

本项目的总体结构可以分为外层和内层结构。外层为 QT 的各类接口与控

件；在内部主要由三个 c++类组成：NFAGraph、DFAGraph 和 LexAnalyzer 类。

1.4 功能分配

程序外层主要负责与用户的交互、获取外部的输入和显示内部输出的数据；程序内层的三个类负责对接受的数据进行分析，并传递分析数据。

1.5 模块外部设计

1.5.1 模块名称和描述

程序分为三个模块：输入模块、分析模块和展示模块。

输入模块：用户在编辑框内输入正则表达式和源程序，然后点击分析，若输入无误，数据会向分析模块传递。

分析模块：当从输入模块获取数据之后，分析模块会进行分析并产生分析结果，并向展示模块传递。

展示模块：从分析模块获取数据后，在展示模块会以表格和文本两种形式展示分析结果。

图 2　展示模块

1.5.2 输入与输出

输入的数据为统一为文本类型。输入正则表达式时，一个表达式占据一行。左侧是正则表达式的名称，然后是一个空格+一个“=”再加上一个空格，即“ = ”，紧接着右侧是表达式的内容。

以下是正则表达式的一个格式示例：

number = DIGITDIGIT*

表达式支持的表达式有：连接（不使用符号）、选择（|）、可选（?）、闭包 （*）和圆括号。支持对这四种符号使用转义字符，如若想将符号“*”表示为表达式的一个转换而不是表达式的运算，则可以用“*”表示。对于输入的源程序，则没有具体的格式要求。

程序的输出，对于图型信息，以表格形式展示，表格横纵坐标描述图的节点和边的信息；文本型信息，则直接展示在编辑框控件中。

1.5.3 界面设计

程序界面可分为左右部分。其中左侧是输入模块，主要提供给用户进行输入、保存等操作；右侧是展示模块，用于查看程序的分析结果，负责展示内容。

图 3　程序界面

1.5.4 错误处理

输入模块的错误处理主要针对用户的不规范输入。例如，当用户空输入或者未设置正则表达式而分析源程序时，程序会出现窗口提示。

在分析模块中，应对错误的主要方式在于，当数据不满足当前分析所需要的所有条件时，程序立刻停止分析。如分析后缀表达式时分析栈为空，则函数立刻返回等等。

图 4　错误提示

二、数据结构设计

2.1 逻辑结构设计

本程序的逻辑结构设计如下图所示：

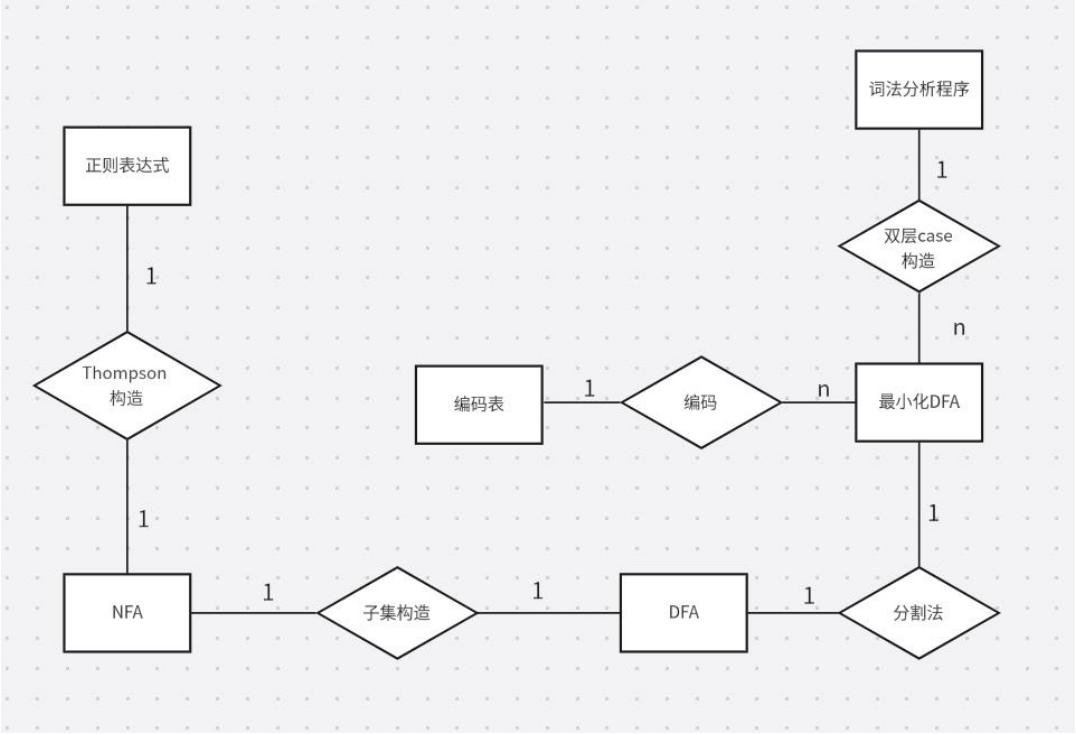

图 5　程序 E-R 图

2.2 物理结构设计

由于本项目的数据量并不大，且所有正则表达式仅解释一个高级程序语言，并无顺序关系，所以直接采取顺序存取方法，且只需要存储实体（多个 C++ 对象），不需要存储联系，容器采用 vector。访问时，直接通过下标获取对应位置的数据。

2.3 数据结构与程序的关系

在用户输入正则表达式后，经过分析，程序生成一条或多条 NFA、DFA 和最小化的 DFA，这些对象用数据结构 vector 存储，届时用户通过随机访问方式选中某条表达式，通过下标直接访问 vector 对应位置，该内容将由分析模块传递到输出模块，并以图表方式展示出来。

三、程序描述

在以下内容中将以类为单位来描述。

3.1 功能

3.1.1 NFAGraph

该类的功能有：接受一个正则表达式并将其转换为后缀、判断该表达式对应单词是否需要解释、用 Thompson 构造法构造 NFA 等。其中单词解释表示某个单词在转换为编码后还需要具体指出其值，比如若 x 是一个标识符，则扫描到 x 时不仅需要输出标识符的编码，还需解释此处的标识符指的是 x，而表示加法的“+”之类的运算符，由于内容是固定不变的，因此无需解释。

3.1.2 DFAGraph

该类的功能有：接受一个 NFAGraph 类对象并用子集构造法转换为 DFA，使用分割法最小化 DFA 等。

3.1.3 LexAnalyzer

该类包含了一组 NFAGraph 对象和一组 DFAGraph 对象。接受一组表达式的字符串，通过初始化方法可以生成上述的两个类的两组对象，并为每个表达式或关键字赋予一个编码。同时，该类还有生成单词编码表和生成词法分析程序的方法。

3.2 输入项目

NFAGraph 的输入项目为一个正则表达式；DFAGraph 的输入项目为一个 NFAGraph 对象；LexAnalyzer 类的输入项目为一组正则表达式。

3.3 输出项目

NFAGraph 的输出项目有表达式内容（去除转义字符）、是否解释、初始状态号、接受状态号、状态转换表和字符集合等等；DFAGraph 的输出项目有初始状态号、接受状态号、字符集合、转换邻接表和自身最小化后的 DFA 对象等等；LexAnalyzer 类的输出项目为一组 NFAGraph 对象、一组 DFAGraph 对象、编码表和词法分析程序等等。

3.4 算法

各模块所采用的算法可用下图表示。

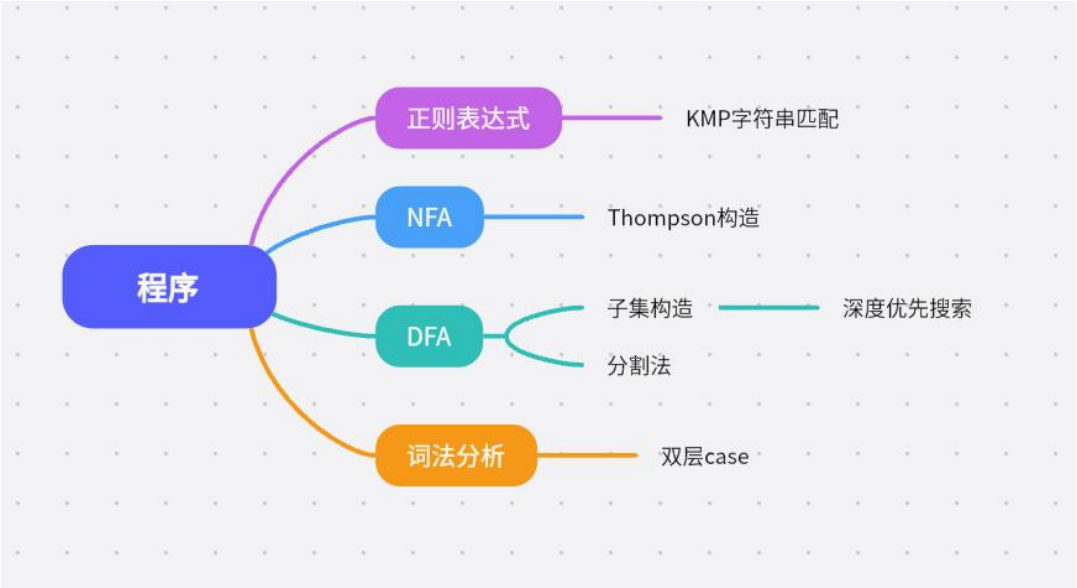

图 6　各模块算法

3.5 函数调用关系

从程序整体上来看，主要功能的函数调用关系如下图所示：

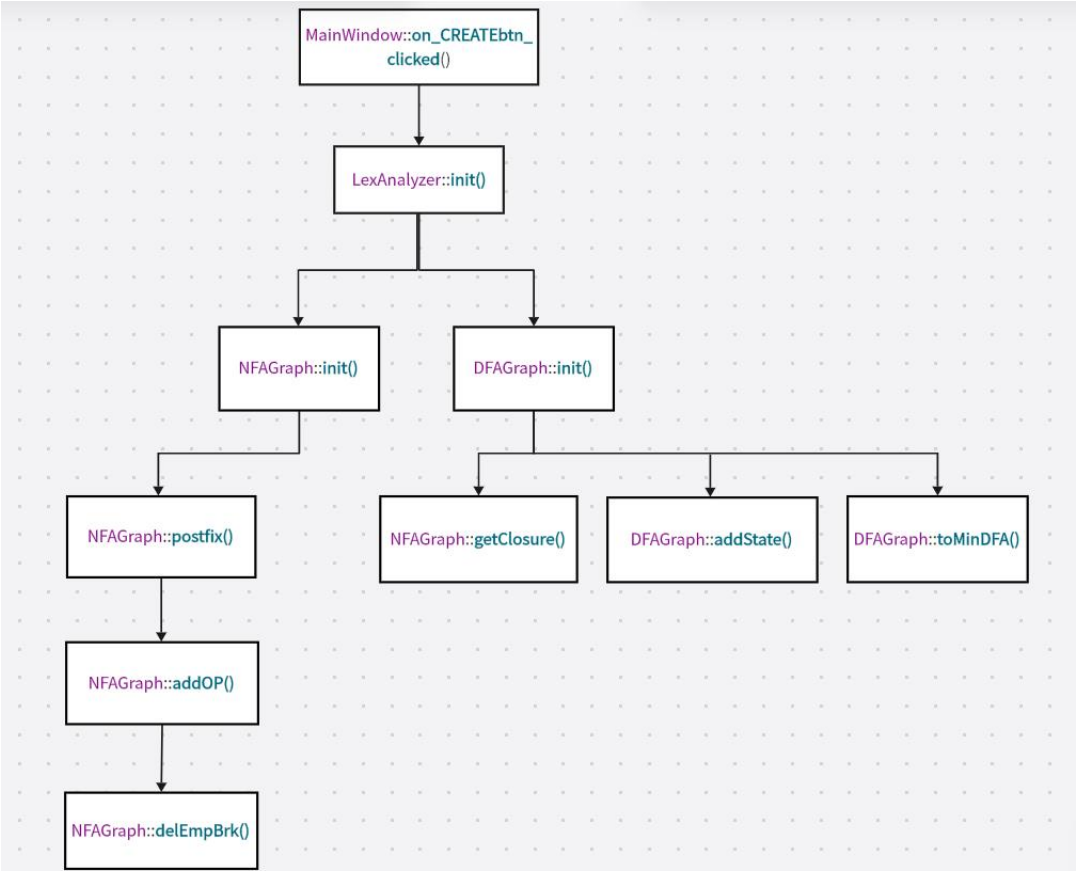

图 7　函数调用关系

3.6 程序逻辑

3.6.1 NFAGraph

将表达式转换为 NFA 采用了 Thompson 构造法。

对于正则表达式的运算，这里使用了转换后缀表达式再运算的方法。由于正则表达式中连接符号为隐式显示，不利于转换后的计算，这里以“&”表示连接，我们需要先为表达式显式添加“&”符号，如 ab 转换为 a&b。添加方法只需对连接运算可能出现的情况进行枚举即可。

参考一般数值计算的中缀转后缀的算法，需要借助数据结构栈实现。接下来的关键问题便是如何设计各运算符的优先级。设置栈内优先数 isp 和栈外优

先数 icp，在数值计算转换后缀表达式中，优先数设计有以下特点：

①乘、除、取余等在中缀表达式具有更高优先级的符号，其栈内外后缀优先数都大于加、减符号；

②优先数相等的只有这几种情况：

栈内“(”遇到栈外“)”，“)”是不入栈的，并且将使相匹配的“(”出栈，因为 icp[‘)’]的优先级极小，在“(”出栈前这两个半括号间的运算符将全部出栈，体现优先运算括号内容的规则；

栈内“)”遇到栈外“(”，这种情况并不会发生，因为 icp[‘)’]的优先级极小，只有“#”的更小，但“#”是栈中第一个元素或表达式终止符，“)”入栈说明表达式有问题。

③“(”入栈后优先数变极低，方便后续运算符入栈。

④除了括号外，对于同个运算符，都有 isp>icp。

在正则表达式运算中，优先级为 *=? > & > |，根据上述特点，可以推知它们的优先数应有如下关系：

$$(^{(外)},)^{(内)} > *,?^{(内)} > *,?^{(外)} > \&^{(内)} > \&^{(外)} > |^{(内)} > |^{(外)} > (^{(内)},)^{(外)}$$

易知可将优先数设计如下：

运算符	栈内（isp）	栈外（icp）
(	0	7
)	7	0
*,?	6	5
&	4	3
|	2	1

后缀表达式转换的步骤如下：

创建空字符串 str，对中缀表达式从左到右检索。遇到字符直接并入 str，若遇到操作符：

①栈空时运到运算符直接入栈；

②当前运算符栈外优先数大于栈顶运算符栈内优先数，当前操作符入栈；

③若小于，栈顶运算符退栈且并入 str；

④若等于，栈顶运算符退栈但不并入，如果退‘(’，读入下一字符；

⑤扫码结束 str 即为后缀表达式。

核心代码实现如下：

```
char ch=s.top();
//①当前操作符栈外优先数大于栈顶操作符栈内优先数，当前操作符入栈；
if(icp[c]>isp[ch])
{
        s.push(c);
        i++;
}
//②若小于，栈顶操作符退栈并输出（或并入储存后缀表达式的字符串）；
else if(icp[c]<isp[ch])
{
        while(!s.empty()&&icp[c]<isp[s.top()])
        {
                str+=s.top();
                s.pop();
        }
}
//③若等于，栈顶操作符退栈但不输出，如果退‘(’，读入下一字符。
else
{
         if(s.top()=='(') i++;
         s.pop();
}
```

现在来考虑如何由得到的后缀表达式转换 NFA。首先我们需要有能够保存 NFA 结构的方法。考虑到使用 Thompson 结构构建 NFA，可以按存储图的方法，使用邻接表保存：

其中 vector 的下标表示了一个状态，每个邻接表中存储的是结构体 edge，表示转换类型和转换后的状态，如 vector[1]里存储了一个 edge(‘a’,2)表示状态 1 能通过 a 转换到状态 2。

可以存储 NFA 结构后，就是考虑如何转换。NFA 类创建一个初始化方法，接受一个后缀正则表达式，准备一个存储 NFA 结构的<初始状态,接受状态>的栈，然后从左向右扫描，根据遇到的字符或运算符按对应的 Thompson 结构，从栈中取出相应数量 NFA，建立新 NFA，随后压入栈。Thompson 中有如下几种结构：

基本转换：

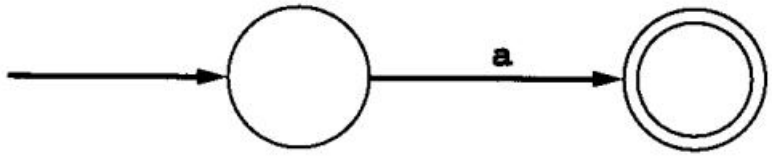

图 8　基本转换的 Thompson 结构

即遇到单个字符时，我们需要创建两个新状态 start 和 end，在 start 的邻接表中记录转换 edge(‘a’,end)，随后压入栈。

闭包*：

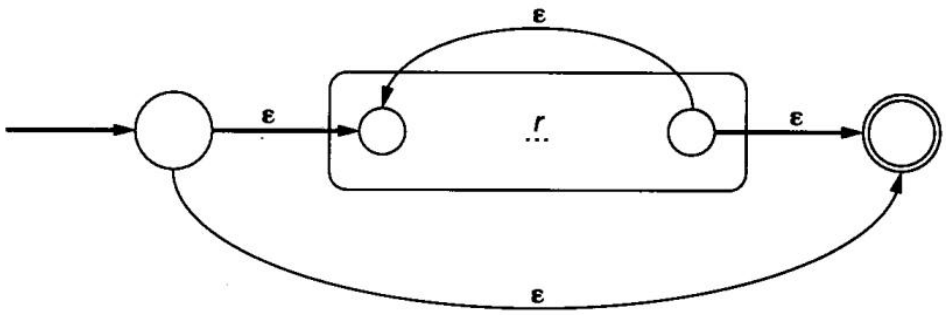

图 9　闭包运算的 Thompson 结构

闭包是单目运算符，需要从栈中取一个 NFA，然后建立两个新状态 start，end，按照 Thomson 分别在邻接表中记录新增的 ε -转换，其中 start 是初始状态，end 是接受状态。

对于连接&、选择 | 等双目运算符，我们需要从栈中取出两个 NFA 建立新的 ε -转换，将运算后的<初始状态,接受状态>入栈，这里不再赘述。

扫描完成后，栈中会剩下一个 NFA 结构即最终结果。我们知道，NFA 有初始状态和接受状态，此时我们如何知道是哪个状态？根据这几个运算符的 Thompson 结构可知，每次运算得到的 NFA 都只有一个初始状态和一个接受状态，也就是说最终的初始状态和接受状态，正好分别对应最后的 NFA 在栈中的存储的值。最后，在计算过程中，可以为类添加一个 set<char>类型成员变量存储出现过的转换类型。

部分核心代码如下：

```
else if(cur=='*')//闭包
```

```
{
        epl=true;
        pii ns=ss.top();
        ss.pop();
        int nsleft=ns.first,nsright=ns.second;
        int start=++NumOfStatus,end=++NumOfStatus;
        graph.push_back(set<edge>()),graph.push_back(set<edge>());
        graph[start].insert(edge('#',end));
        graph[start].insert(edge('#',nsleft));
        graph[nsright].insert(edge('#',end));
        graph[nsright].insert(edge('#',nsleft));
        ss.push({start,end});
        cs.insert('#');
}
```

3.6.2 DFAGraph

考虑下如何从 NFA 转换到 DFA。这里采用子集构造的方法。首先考虑下 DFA 类如何存储。DFA 和 NFA 结构相似，只是没有 ε -转换，所以可以用同样类型的邻接表和字符表存储。需要注意的是 DFA 可能存在多个接受状态，所以我们需要使用 vector 容器去标注各状态是否是接受状态了。此外，由 NFA 转换的 DFA，其每个状态可能包含 NFA 中的多个状态，所以还需要建立 vector<set<int>>类型由状态编号到元素集合的映射。

使用子集构造的方法来进行 NFA 到 DFA 的转换。简要地说步骤有以下几点：

（1）列出 NFA 各状态的 ε -闭包；

（2）找出 NFA 中的所有非 ε -转换；

（3）以 NFA 初始状态的 ε -闭包作为 DFA 的初始状态，以（2）中找到的规则进行转换，若产生新状态，对新状态继续转换，还要用之前的搜索过的转换对新状态进行转换，直到不再产生新的状态。

（4）若状态存在中有元素是 NFA 中的接受状态，则该状态也是 DFA 的接受状态。

这个方法需要我们获取 NFA 中各状态的 ε -闭包。于是为 NFA 添加成员函数 getClosure 获取 ε -闭包。由于 ε -闭包是某一状态 i 经零个、一个或多个 ε -转换能得到的集合，容易想到使用深度搜索的方法在 NFA 结构中沿着 ε -

转换搜索：

```
void NFAGraph::DFS(set<int> &s,vector<bool> &f,int fa)//搜索到的闭包集合，是否访问，父节点编号
{
    s.insert(fa);
    f[fa]=true;
    for(auto node:graph[fa])
    {
        if(node.CvType=='#')//找到ε转换
            DFS(s,f,node.vj);
    }
}
```

现在我们有ε-闭包了，需要设计一个算法使得 DFA 产生的每个新状态都进行所有转换。可以考虑双重循环：外层是 DFA 的状态集 vector<set<int>>类型成员变量 states，内层是字符集，这样，每次产生的新状态会插入 states 尾部，使得循环继续进行；进入内循环则可以对当前状态遍历所有转换。在这之前，先创建 map<char,map<int,int>>类型字符转换表，方便搜索在某个转换中哪些状态转换到了自身或其他状态。在第 i 次循环中，首先遍历 states[i]中元素，将元素在字符转换表中搜索，看看是否有转换，有则保留转换后对应的ε-闭包集合。将所有这样的闭包集合并运算得到状态，在 states 中搜索该状态是否已存在，若不存在则尾插建立新状态。在这过程中产生了 DFA 中状态的转换，可以顺便建立 DFA 的邻接表。

代码：

```
void DFAGraph::init(NFAGraph &nfa)
{
    isMinimized=false;
    vector<set<int>> closure=nfa.getClosure();//ε-闭包
    map<char,map<int,int>> chtrans;//字符转换表
    set<char> chs=nfa.getCharSet();//获取字符集
    for(auto c:chs)
    {
        if(c!='#')
        {
            cs.insert(c);
            chtrans[c]=nfa.getCharTrans(c);
        }
    }
```

```
    if(closure.size()<=1)
        return;

    states.push_back(set<int>());
    isAccepted.push_back(false);

    //加入初始状态
    int initst=nfa.getInitState();
    states.push_back(closure[initst]);
    graph.push_back(set<edge>());//下标 0 不用
    graph.push_back(set<edge>());//为初始状态创建邻接表
    InitState=states.size()-1;

closure[initst].find(nfa.getAcceptState())!=closure[initst].end()?isAccepted
.push_back(true):isAccepted.push_back(false);
    //开始转换
    for(int i=1;i<states.size();i++)
    {
        for(mcmi mi=chtrans.begin();mi!=chtrans.end();mi++)
        {
            char ch=mi->first;//当前转换字符
            set<int> res;
            for(auto j:states[i])//遍历 i 的状态集合
            {
                map<int,int>::iterator it=chtrans[ch].find(j);
                if(it!=chtrans[ch].end())//当前字符 ch，当前状态 j，若找到转换
的状态，进行集合并

set_union(res.begin(),res.end(),closure[it->second].begin(),closure[it->seco
nd].end(),inserter(res,res.begin()));
            }
            if(res.empty())
                continue;

            int sz=states.size();
            int w=addState(nfa,res);
            if(sz<states.size())//若 size 发生变化，说明产生新状态
                graph.push_back(set<edge> ());
            graph[i].insert(edge(ch,w));
        }
    }
    NumOfStates=states.size()-1;
}
```

然后，再思考如何实现分割方法对 DFA 执行最小化。最小化 DFA 结构和

DFA 相同，所以可以用同个类表示，同时 DFA 类中应该添加一个最小化自身的成员函数。

先来研究如何为模拟过程做准备。首先我们需要知道集合最后被如何分割，创建 vector<int>型集合号映射表，它为每个状态赋予一个编号，同个编号的状态表示划分到同个集合中。然后考虑如何持续进行分割直到算法结束。设计一个双端队列存储待分割的集合序列，每次弹出一个集合进行分割，最终需要判断当前集合是否被分割了，若进行了分割，则将多于一个元素的集合继续加在双端队列末端，仅一个元素的集合直接存储在我们的最终结果中，且产生了新的分割集合后，之前的集合可能又可以分割，于是应该再运行一遍算法；若未被分割，说明集合内各状态是不可分割的，可加入最终结果而无需加入队列。这使得我们的算法能够自动停止（队列为空且没有发生分割）并得到正确答案。初始时，将状态按接受状态和非接受状态划分，加入队列。

接下来考虑如何分割。在分割时，一个集合可能被分割为多个集合，用 vector<set<int>>收集分割的集合。由于我们需要对比集合内各状态的转换是否是等价，可以用 vector<set<edge>>创建转移集合表，记录一个状态经过哪些转换到达哪些集合。先从邻接表和集合号映射表中获取 edge(转换名,集合号)内容，然后比对与集合内其他状态的转移集合是否相同，若不同，则可区分，将该状态分割到一个新集合中。将具有相同转移集合表的状态收集为各个集合。若一个集合被分割了，我们需要创建新的集合号给集合映射表中被分割出去的状态。

循环结束之后，我们已经可以从集合号映射表中知道哪些状态被划分到同一集合，可以进行合并了。为最小化的 DFA 变量创建集合数量大小的邻接表。首先需要知道哪些合并后哪些集合是接受状态，这只需要从 DFA 的接受状态表中映射到相应集合中，将接受状态所在的集合修改为接受状态即可。其次便是构建最小化 DFA 的邻接表，同样将状态到状态的转换映射为集合到集合的转换即可。以各个集合便为作为最小化 DFA 的各个状态，如此便完成了最小化 DFA 最困难的部分。

核心代码：

```
bool part=true;//当上一次队列中集合发生过划分时，需要重新检查一遍是否可划分
while(part)
```

```
{
    if(mNumStates>2)//统计各集合划分情况，对多于一个元素的集合检查是否能划分
    {
          for(int ii=1;ii<=mNumStates;ii++)
           {
                set<int> sss;
                for(int ij=1;ij<=isAccepted.size();ij++)
                    if(setNum[ij]==ii)
                        sss.insert(ij);
                if(sss.size()>1)
                    dq.push_back(sss);
           }
        }
        part=false;
        while(!dq.empty())//开始最小化
        {
            set<int> s=dq.front();
            dq.pop_front();
            vector<set<edge>> ve;//转移集合表
            vector<set<int>> vs;//划分的集合
            for(auto i:s)//对集合 s 进行分割
            {
                //获取 i 的能转移到的集合号的集合
                set<edge> ste;
                for(int vi=1;vi<graph.size();vi++)
                {
                    for(auto cm:graph[i])//遍历 i 的邻接表
                        ste.insert(edge(cm.CvType,setNum[cm.vj]));//在集合号
表中找到转换后状态所在的集合
                }

                if(vs.size()==0)
                {
                    ve.push_back(ste);
                    vs.push_back(set<int>());
                    vs[0].insert(i);
                }
                else
                {
                    bool fequal=false;//转换相同标志
                    for(int vi=0;vi<ve.size();vi++)
                    {
                        if(transEqual(ve[vi],ste))
                        {
```

```
                            vs[vi].insert(i);//转换相同
                            fequal=true;
                            break;
                        }
                    }
                    if(!fequal)//不同，划分
                    {
                        ve.push_back(ste);
                        vs.push_back(set<int>());
                        vs[vs.size()-1].insert(i);
                        part=true;
                    }
                }
            }
        //清算划分集合 vs
        for(int j=1;j<vs.size();j++)//为划分出去的集合编号
        {
            if(vs[j].size()>1)
               dq.push_back(vs[j]);
            mNumStates++;
           for(auto vk:vs[j])//遍历集合内状态，为其赋予新的状态号
           {
               setNum[vk]=mNumStates;
           }
        }
    }
}
```

3.6.3 LexAnalyzer

到这里我们已经完成了从正则表达式到最小化 DFA 的转换。现在需要思考的是如何统合一个程序语言的多个正则表达式。首先，该类的成员变量有使用 vector 存储的正则表达式内容 REXcontents、表达式名 regexName、一组 NFAGraph 对象和一组 DFAGraph 对象。在接受一组字符串后，分别调用各个类的初始化函数进行初始化。对于一般的正则表达式，该类会使用 map 为其映射到一个编码。对于关键字，使用特殊命名识别的方法将其存到 map 结构的映射表并直接编号。对于注释的正则表达式，由于与其他表达式在匹配方法上的不同，我们也另外存储。

对于某条正则表达式，由于它可能与其他的正则表达式是上下级关系，在这里想出了两种办法处理：

①在表达式之间采用嵌套关系。如表达式：

identifier = LETTER(LETTER|DIGIT)*

可表示为 identifier 与 LETTER 和 DIGIT 的嵌套关系，届时转换可以直接用正则表达式名表示。如上述表达式可用下图表示：

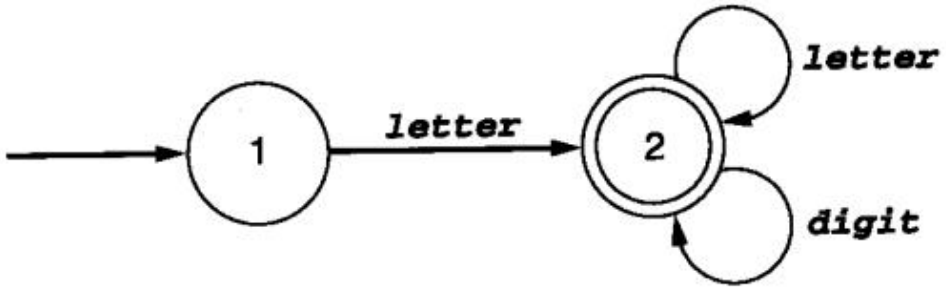

图 10　TINY 标识符的 DFA 图

在词法分析时，体现了表达式的嵌套关系：先进入 letter 的匹配，若成功，判断可能为标识符，再进行下一步匹配。

②在读入字符串后将表达式直接在内部展开，即用表达式内容替换表达式名。这样就将表达式嵌套问题简化成了单字符的转换。缺点是这可能使表达式的 NFA、DFA 状态数量变得庞大且不直观。考虑到工作量与实现难度，这里我采用第②种方法实现。在读入所有表达式之后，采用 KMP 字符串匹配算法执行匹配，并替换表达式内容，然后再调用 NFA 的初始化。

代码实现如下：

```
//执行 KMP 算法，匹配并替换正则表达式
vector<int> next[NumOfRegex];
for(int k=0;k<NumOfRegex;k++)
{
    next[k].resize(regexName[k].length());
    getKMPNext(next[k],k);
}

//执行 KMP 匹配并替换表达式
for(int i=0;i<NumOfRegex;i++)
{
    for(int j=0;j<NumOfRegex;j++)
    {
        if(i==j)
            continue;
        string s=REXcontents[i],t=regexName[j];
```

```
            int sl=s.length(),tl=t.length();
            if(sl<tl)
                continue;
            int ii=0,ij=0;
            while(ii<sl&&ij<tl)
            {
                if(ij==-1||s[ii]==t[ij])
                    ii++,ij++;
                else
                    ij=next[j][ij];
                if(ij>=tl)
                {
                    string ss="("+REXcontents[j];
                    ss+=")";
                    s=s.substr(0,ii-tl)+ss+s.substr(ii,sl-ii);
                    ij=0;
                    ii=ii-tl+ss.length();
                    sl=s.length();
                }
            }
            REXcontents[i]=s;
        }
```

在词法分析并输出编码阶段，注释的 DFA 匹配与其他略有不同。其他 DFA 是匹配成功后即可输出编码，而注释类型无需进行词法分析，应该被跳过，所以先在正则表达式上进行命名特殊化：用“L_COMMENT”表示注释的起始和用“R_COMMENT”表示注释的结束。注释先于其他 DFA 进行匹配。当某匹配到某个“L_COMMENT”时，不断移动指针直到成功匹配配对的“R_COMMENT”，否则输出错误信息，中间的字符都被略过。其他 DFA 匹配成功后，还得执行一次搜索是否是关键字。

我们注意到 NFAGraph 中存在一个解释标记变量 epl，它决定了在输出编码后，是否还要输出（解释）单词的具体内容。判断是否解释的方法在于正则表达式对应的内容形式是否是固定的，即若表达式不是单字符且执行了连接以外的运算，则表达式有不止一种可以被匹配的形式，于是需要标记 epl 为 true。

生成词法分析程序的方法依旧采用双层 case 方法。核心是将每个 DFA 生成的双层 case 都包装到一个函数中，通过函数指针数组组织起来，在主函数中生成一样的匹配方法，不同的是调用 DFAGraph 的邻接表在词法分析程序中变为了调用含有双层 case 匹配的函数。

任务二

一、总体设计

1.1 需求概述

本项目要求设计一个分析程序，针对某一高级程序设计语言的 BNF 文法，可以分析并生成相应的 First 集合、Follow 集合、LR(0)的 DFA 和 SLR(1)分析表，并能根据项目一输出的编码表和编码后的源程序，进行语法分析并建立语法树。

1.2 处理流程

本项目的处理流程可以用以下流程图表示：

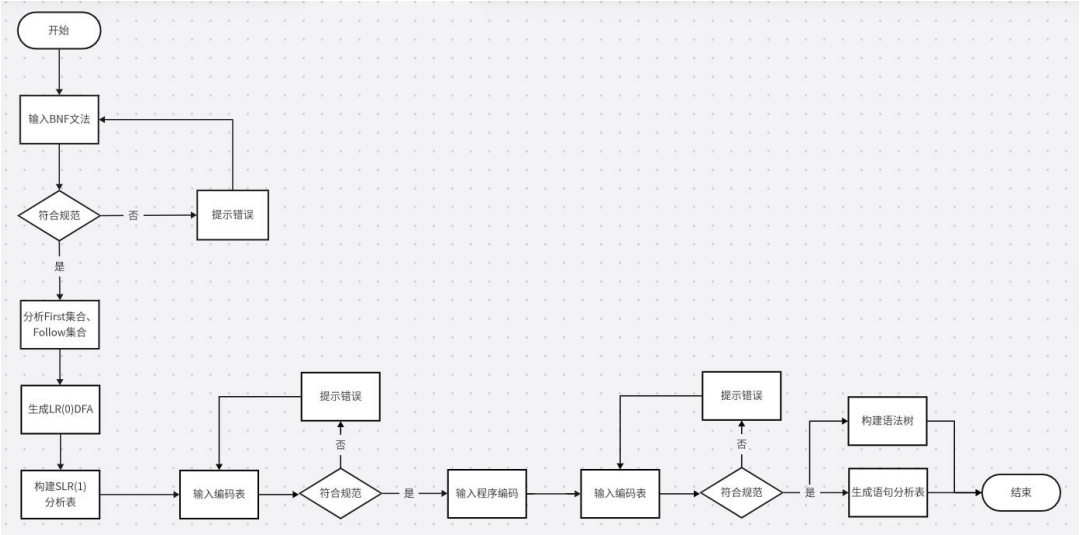

图 11　处理流程图

1.3 总体结构和模块外部设计

本项目的总体结构可以分为外层和内层结构。外层为 QT 的各类接口与控件；在内部主要由一个 C++类 slrItem 完成工作。

1.4 功能分配

程序外层主要负责与用户的交互、获取外部的输入和显示内部输出的数据；程序内层的 slrItem 类负责对接受的数据进行分析，并向外层传递分析数据。

1.5 模块外部设计

1.5.1 模块名称和描述

程序分为三个模块：输入模块、分析模块和展示模块。

输入模块：用户在编辑框内输入 BNF 文法、编码表和源程序编码，然后点击分析，若输入规范，数据会向分析模块传递。

分析模块：当从输入模块获取数据之后，分析模块会进行分析并产生分析结果，并向展示模块传递。

展示模块：从分析模块获取数据后，在展示模块会以表格形式展示分析结果。

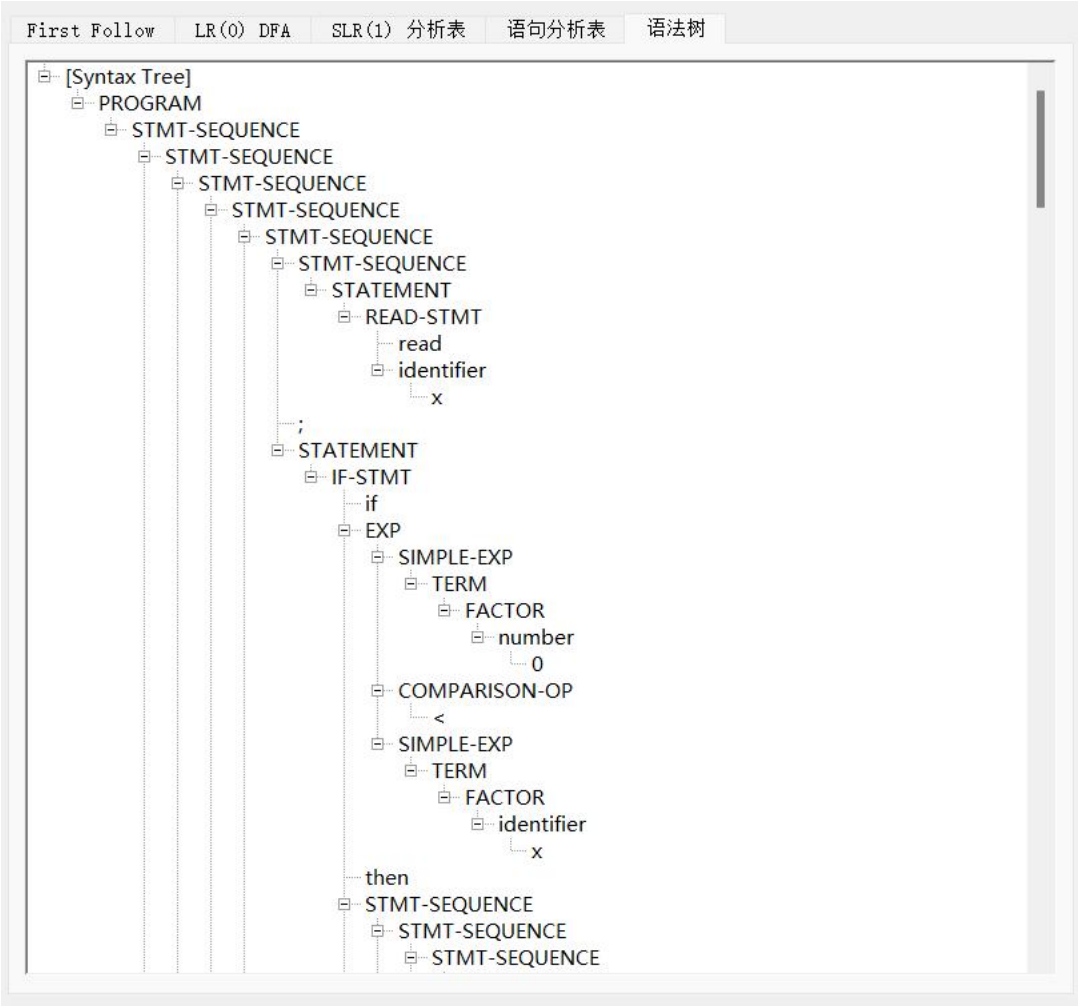

图 12　展示模块

1.5.2 输入与输出

对于每一条文法，左侧是非终结符，然后用一个空格+“->”+一个空格隔开，即“ -> ”，右侧是文法规则的内容。每一个符号之间都需要使用空格分隔。文法支持使用“|”符号表示“或”。如下是一条文法规则的输入示例：

STMT-SEQUENCE -> STMT-SEQUENCE ; STATEMENT | STATEMENT

程序的输出，对于图型信息，以表格形式展示，表格横纵坐标描述图的节点和边的信息以及节点的具体内容；构造的语法树则直接以树形结构展示。

1.5.3 界面设计

程序界面可分为左右部分。其中左侧是输入模块，主要提供给用户进行输入、保存等操作；右侧是展示模块，用于查看程序的分析结果，负责展示内容。

图 13　程序界面

1.5.4 错误处理

输入模块的错误处理主要针对用户的不规范输入。例如，当用户空输入或者未分析文法和设置编码表而分析源程序编码，程序会出现窗口提示。

在分析模块中，应对错误的主要方式在于，当数据不满足当前分析所需要的所有条件时，程序立刻停止分析。如分析时分析栈为空，则函数立刻返回等。

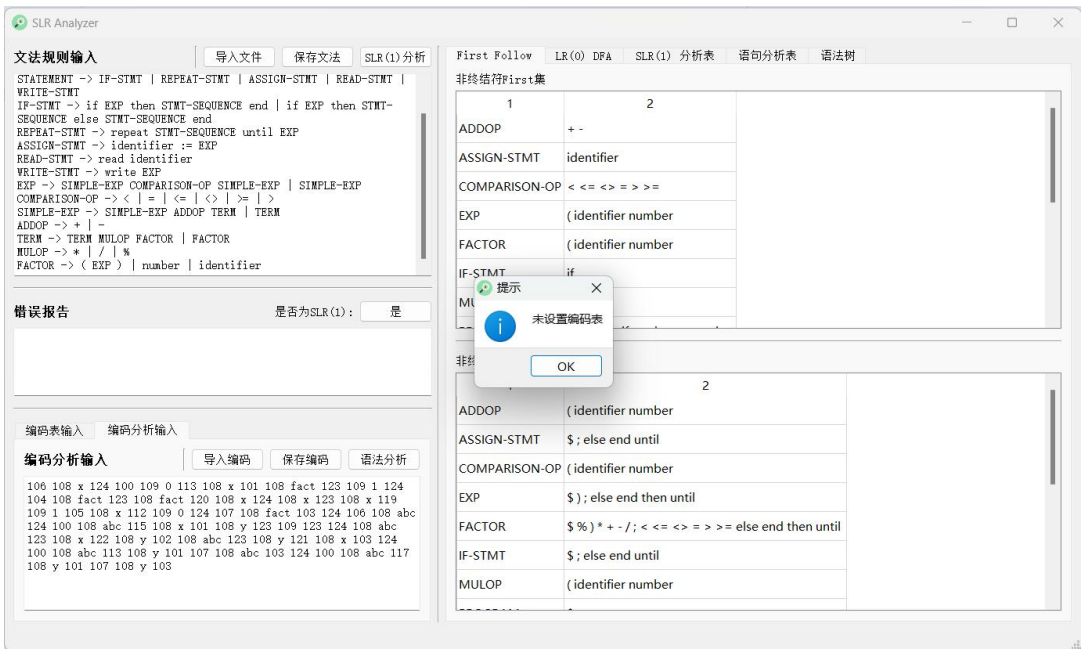

图 14　错误提示

二、数据结构设计

2.1 逻辑结构设计

本程序的逻辑结构设计如下图所示：

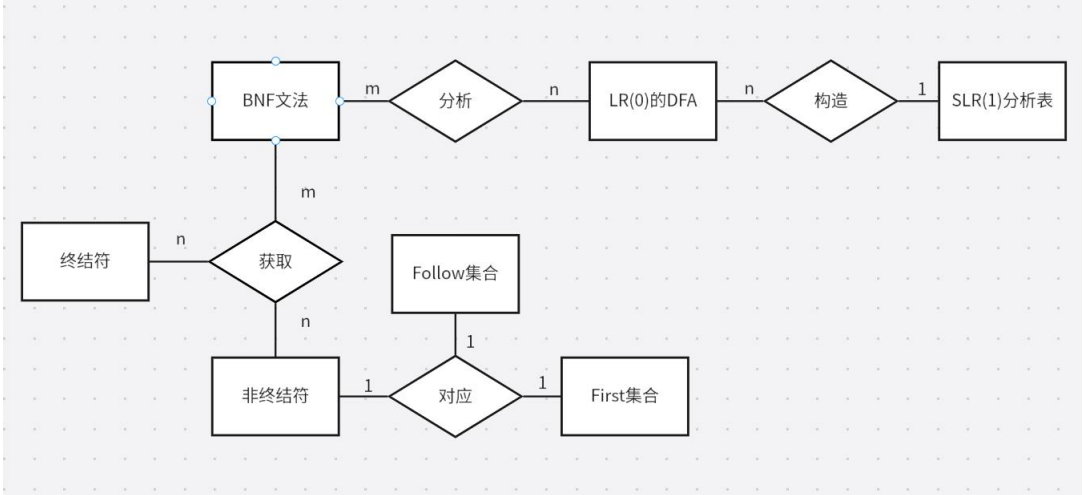

图 15　程序 E-R 图

2.2 物理结构设计

由于本项目的数据量并不大，且所有文法规则仅解释一个高级程序语言，所以直接采取顺序存取方法，且只需要存储实体（基本类型变量和结构体），

不需要存储联系，容器主要采用 vector 和 map。访问时，直接通过下标或键获取对应位置的数据。

2.3 数据结构与程序的关系

由于文法规则的格式是一个非终结符对应一条规则，所以很容易想到使用键值对来存储文法。而每个非终结符与其 First 集合和 Follow 集合之间是一对多的关系，于是容易想到把非终结符内容作为键，集合类型 set 作为值，在搜索时可以直接通过键来定位；而 LR(0)的 DFA 与 SLR(1)分析表都有序号排列，所以可以通过 vector 存储，分析时可借助 push_back 方法不断创建新状态。

三、程序描述

3.1 功能

slrItem 对象可以读入所有文法规则，通过初始化 init 方法对文法进行分析，求得各非终结符的 First 集合和 Follow 集合，然后再求得 LR(0)的 DFA，并构建 SLR(1)表的移进项和归约项；之后可以通过公有成员方法 analysis 进行语句分析并返回一个语法树节点。还能根据编码表对一个程序编码进行解码和分析。

3.2 输入项目

slrItem 对象的 init 方法接受从 QT 控件 QTextWidget 传来的多行文法规则发字符串；getCodeList 方法接受字符串类型的编码表；analysis 方法接受一个内容为项目一输出的程序编码的字符串，并需要一个指向大小至少为 4 的 vector<string>类型的指针。

3.3 输出项目

slrItem 的输出项目有：set<string>类型的 First 集合和 Follow 集合、vector<vector<item>>类型的 LR(0)项目、使用 map 存储的移进项和归约项存储在 vector<string>[4]中的 SLR(1)分析表和语法树的根节点等。

3.4 算法

程序中各项目的分析算法主要是模拟，通过直接模拟手工分析的方法来求得结果。在 SLR(1)语句分析中，同样使用双栈法去模拟分析过程。

3.5 函数调用关系

程序的主要功能的函数调用关系可以表示为下图：

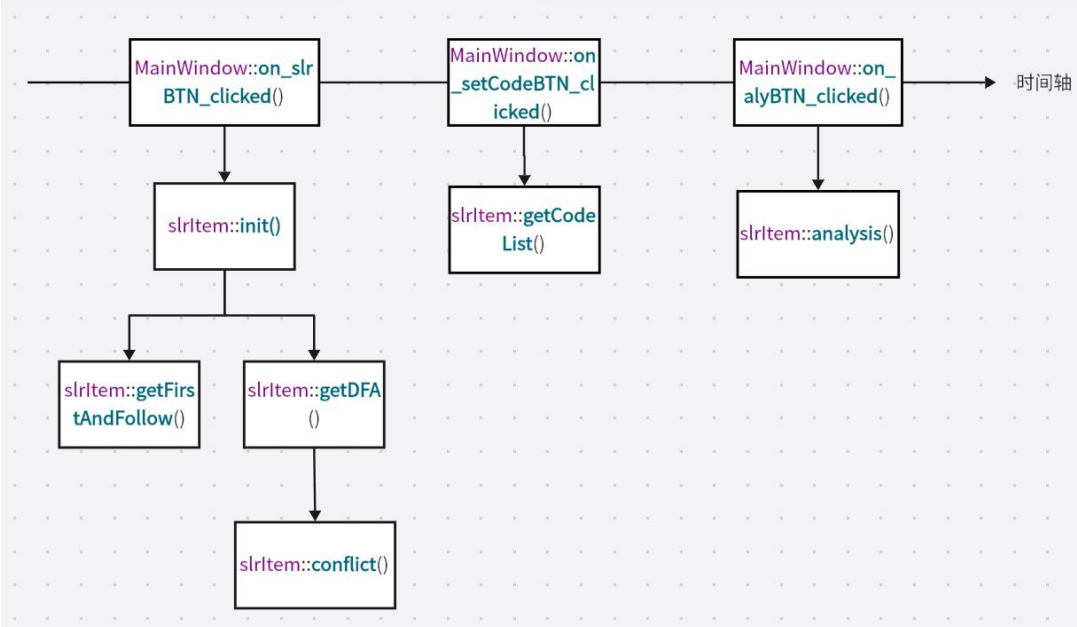

图 16　函数调用关系

3.6 程序逻辑

3.6.1 扩充文法

如果一个非终结符的开始符号有多个产生式，那我们就要在开始文法分析前扩充文法。程序是这样实现的：采用遍历搜索的方法检查起始符号是否出现了多次，若出现多次，则需要在该起始符号后添加“’”符号作为新的起始符号。

3.6.2 求解非终结符 First 集合和 Follow 集合

求所有符号的 First 集的求法可以描述如下：初始化所有符号对应的 First 集合为空，然后执行循环：遍历文法规则的产生式，定义产生式对应的 key 和 value，其中 key 为产生式的左部非终结符，value 为产生式的右部规则；遍历右部符号序列中的每个符号，对于每个符号，为它创建 First 集合 set；将 set

中的"ε"（程序内表示为字符串“empty”）去除，然后将结果加入左部符号对应的 First 集合，如果 set 中不包含"ε"，则停止对右部符号的处理，如果处理完右部符号序列后，k 等于 value 的大小，说明右部符号序列中的所有符号都可以推导出空串"ε"，则将{"ε"}加入左部符号对应的 First 集合；若所有符号的 First 集合不再发生改变，循环停止。

通过这个算法，可以计算出文法规则中每个非终结符的 First 集合，从而用于进一步求出 Follow 集。将非终结符筛选出来，将其 First 集显示在界面。

非终结符的 Follow 集求解算法可以描述如下：初始化所有非终结符对应的 Follow 集合为空，然后进行循环：遍历文法规则的产生式，产生式对应 key 和 value，其中 key 为产生式的左部非终结符，value 为产生式的右部符号序列，遍历右部符号序列中的每个符号，如果符号是非终结符，则将该非终结符后面的符号序列的 First 集合加入该非终结符的 Follow 集合；如果该非终结符后面的符号序列的 First 集合中包含空串"ε"，则将产生式左部非终结符的 Follow 集合加入该非终结符的 Follow 集合，直到所有非终结符的 Follow 集合不发生改变时，循环停止。

代码实现如下：

```
//求非终结符的 first 集
for(auto &nt: this->NonTerSet)
    nTerfollow[nt] = set<string>();
bool isChange = true;
while(isChange)
{
    isChange = false;
    for(auto &p: this->m)
    {
        string key = p.first;
        vector<vector<string>> values = p.second;
        for(auto &value: values)
        {
            int k;
            for(k=0; k<value.size(); ++k)
            {
                auto && first_set_k = this->getNterFirst(value[k]);
                for(auto &s: first_set_k)
                {
                    if((s!="empty") && (this->nTerfirst[key].find(s) ==
```

```
this->nTerfirst[key].end()))
                        {
                            nTerfirst[key].insert(s);
                            isChange = true;
                        }
                    }
                    if(first_set_k.find("empty") == first_set_k.end())
                    {
                        break;
                    }
                }
                if(k == value.size() and (this->nTerfirst[key].find("empty") ==
this->nTerfirst[key].end()))
                {
                    nTerfirst[key].insert("empty");
                    isChange = true;
                }
            }
        }
    }
    //求非终结符的 follow 集
    for(auto &nt: this->NonTerSet)
        nTerfollow[nt] = set<string>();
    nTerfollow[this->accept].insert("$");
    isChange = true;
    while(isChange)
    {
        isChange = false;
        for(auto &p: this->m)
        { //遍历每一个产生式 p
            string key = p.first;
            vector<vector<string>> values = p.second;
            for(auto &value: values)
            {
                //遍历 value 中每一个非终结符
                for(auto it=value.begin(); it!=value.end(); ++it)
                {
                    if(this->NonTerSet.find(*it) != this->NonTerSet.end())
                    {
                        //特殊情况
                        if(it+1 == value.end())
                        {
                            for(auto &x: this->getFollow(key))
                            {
```

```
                                if(this->nTerfollow[*it].find(x) ==
this->nTerfollow[*it].end())
                                {
                                    this->nTerfollow[*it].insert(x);
                                    isChange = true;
                                }
                            }
                        }
                        else
                        {
                            set<string> first_set =
getFirst(vector<string>(it+1, value.end()));
                            for(auto &x: first_set)
                            {
                                if(this->nTerfollow[*it].find(x) ==
this->nTerfollow[*it].end() and x != "empty")
                                {
                                    this->nTerfollow[*it].insert(x);
                                    isChange = true;
                                }
                            }
                            if(first_set.find("empty") != first_set.end())//
如果有@符号
                            {
                                this->nTerfollow[*it].erase("empty");
                                for(auto &x: this->getFollow(key))
                                {
                                    if(this->nTerfollow[*it].find(x) ==
this->nTerfollow[*it].end())
                                    {
                                        this->nTerfollow[*it].insert(x);
                                        isChange = true;
                                    }
                                }
                            }
                        }

                    }
                }
            }
        }
    }
}
```

3.6.3 LR(0)的 DFA

在 LR(0)文法的 DFA 中，“.”表示了我们正在扫描的位置，当其位于项目末尾时，我们需要进行规约；当期位于项目起始或内部时，需要进行移进，所以可以用标识符区分两种操作，以便后续构建 SLR(1)分析表。

DFA 构建的算法可以描述如下：

初始化项目序列，其中只有一个项目，且项目只有一个项目，该项目为开始符号的初始项目，对于每个项目，初始化其移进类型和归约类型为空字典。

扩展项目中的每个项目，对于项目 j，如果是移进项，则获取转换符号 t 和移进后的新项目 p，如果 t 不在移进类型的关键字集合中，则判断 p 是否已经存在于其他项目中。如果存在，则将 t 和对应的项目加入移进类型；如果不存在，则向项目集合中插入一个含有 p 的新项目，然后将 t 和对应的项目加入移进类型；如果 t 已经在移进类型的关键字集合中，则获取对应的项目 k。如果 p 不在项目 k 中，则将 p 加入项目 k 中。

对于归约项，获取项目 j.key 对应的 follow 集合 follow。

对于 follow 集合中的每个符号 t，如果 t 已经在移进类型中存在，则说明文法不是 SLR(1)文法，因为不满足 SLR(1)项目中归约项.key 的 follow 集合交集为空的原则。否则，将 t 和对应的项目加入移进类型。

代码实现如下：

```
    vector<item> node{item{accept, 0, 0, 1}};
    DFAItem.push_back(node);
    sft_sft = false;
    //循环处理每一个结点 i
    for(int i=0; i<DFAItem.size(); ++i)
    {
        shift[i] = map<string, int>(),reduce[i] = map<string, int>();

        extend(i);

        //遍历该状态的项目
        for(int j=0; j<DFAItem[i].size(); ++j)
        {
            //处理归约项
            item &p = DFAItem[i][j];
            if(p.type == 2){
                set<string> follow_set = this->getFollow(p.key);
                for(auto &s: follow_set)
```

```
                {
                    if(reduce[i].find(s) != reduce[i].end())//找到
                    {
                        sft_sft = true;
                        stringstream ss;
                        ss <<"规约-规约冲突：DFA 的状态"<<i<<"中归约项目的
follow 集的交集不为空\n";
                        alyReport += ss.str();
                    }
                    this->reduce[i][s] = j;
                }
            }
            //处理移进项
            else
            {
                string t = m[p.key][p.value][p.index];
                item np(p.key, p.value, p.index+1, 1);
                if(np.index >= m[p.key][p.value].size())
                    np.type = 2;
                if(this->shift[i].find(t)==shift[i].end())
                {
                    //转换 t 未存在
                    int k=0;
                    for(k=0; k<DFAItem.size(); ++k)
                    {
                        vector<item> & n = DFAItem[k];
                        if(find(n.begin(), n.end(), np) != n.end())
                            break;
                    }
                    if(k < DFAItem.size())//项目已经存在
                    {
                        shift[i][t] = k;
                    }
                    else//项目未存在
                    {
                        //新建节点
                        DFAItem.push_back(vector<item>{np});
                        shift[i][t] = this->DFAItem.size()-1;
                    }
                }
                else//转换 t 存在
                {
                    int k = shift[i][t];
                    vector<item> & n = DFAItem[k];
```

```
                    if(find(n.begin(), n.end(), np) == n.end())//项目不存在
                    {
                        DFAItem[k].push_back(np);
                    }
                }
            }
        }
    }
```

3.6.4 SLR(1)分析表构建

我们会发现在上一个步骤中，除了构建 DFA 外，还顺便进行了规约-规约冲突的检测，这是判断是否为 SLR(1)文法的一部分。当然，这还不够，我们还需要检测是否存在移进-规约冲突：

对于每个项目，如果其移进符号集和归约符号集有交集，则说明文法不是 SLR(1)文法，因为不满足 SLR(1)项目中移进符号不存在于该项目中任何一个归约项 key 的 follow 集合。

得到 DFA 后，由于我们的结构体存储了项目集、转换类型和转换符号，构建 SLR(1)分析表便很简单了，其算法可以描述如下：

从第 0 个 DFA 状态开始，对于状态的每个项目，查询转换符号，并根据转换类型，由 map 追溯到所转换到的状态，填入表中即可。

代码如下：

```
  for(int i=0; i<dfa.size(); ++i)
    {
        ui->SLRWidget->setItem(i, 0, new
QTableWidgetItem(QString::number(i)));
        int col=1;
        for(auto &s: sSet)
        {
            if(shift[i].find(s) != shift[i].end())//移进项
            {
                if(ts.find(s) != ts.end())
                    ui->SLRWidget->setItem(i, col, new
QTableWidgetItem(QString::number(shift[i][s])));
                else
                {
                    string s1="s";
                    s1+=to_string(shift[i][s]);
                    ui->SLRWidget->setItem(i, col, new
```

```
QTableWidgetItem(QString::fromStdString(s1)));
                }
            }
            else if(reduce[i].find(s) != reduce[i].end())//归约项
            {
                item & p = dfa[i][reduce[i][s]];
                if(p.key == slr->getAccept())
                    ui->SLRWidget->setItem(i, col, new QTableWidgetItem("接受
"));
                else
                {
                    string a = "r(";
                    a += p.key;
                    a += " -> ";
                    vector<string> & value = m[p.key][p.value];
                    for(auto &b: value)
                    {
                        a += b;
                        a += " ";
                    }
                    a[a.length()-1] = ')';
                    ui->SLRWidget->setItem(i, col, new
QTableWidgetItem(QString::fromStdString(a)));
                }
            }
            col++;
        }
    }
```

3.6.5 读入编码表

根据项目一输出的编码表，使用字符串扫描的方法，建立编码——词法元素的键值对。若编码数字后面带有一个“*”，表示该词法元素需要进行解释，因此在后续解码编码时，除了读入词法元素的编码，还要继续读入一个紧随其后的 token，这个 token 也就是所解释的内容。

3.6.6 分析句子并生成语法树

使用双栈法，在电子书中已经描述得很清楚了，照着做就行了，因为我们现在已经得到了所需的 SLR(1)分析表。

分析的算法描述如下：

定义状态栈 s1 和输入栈 s2，并进行初始化。s1 的初始值为 0，s2 的值为用户输入的句子，句子中的每个符号都加入到输入栈中，最后加入结束符"$"。

进入循环；

输出当前步骤的编号和状态栈和输入输入栈的内容。

获取状态栈的栈顶元素 f 和输入栈的头元素 s,如果 s 是结点 f 的移进字符，则说明要进行移进操作。

弹出输入栈的头元素，根据 s 找到结点 k，并获取转换后的结点编号 t。

将 s 和 t 依次入栈。

输出"移进 s"，否则，说明要进行归约操作。

根据 f 找到归约项目编号 t，根据 t 得到对应的归约项 p，根据 p 的 key 找到对应的 value，如果 key 是开始符号，则整个语句归约完成，输出"接受"并结束循环；如果 value 的第一个符号不是空串" ε "，则弹出 s1 的 2*value.size 个元素。

更新 f 为栈顶元素，s 为 key，根据 s 找到结点 k，并获取转换后的结点编号 t，将 s 和 t 依次入栈，并输出"归约 key -> value"。

那么，如何生成语法树呢？首先，我们需要定义一个树节点的结构体：

```
struct node//语法树节点
{
    string content;//节点内容
    vector<node> child;//节点孩子
    node(){content="";}
    node(string con) {content=con;}
};
```

每个节点包含一个内容和若干个孩子。内容存储的是终结符号或非终结符号。首先，我们需要将接收的编码根据编码表转换回词法元素。例如，编码表中有内容：

read 106

identifier 108*

而我们现在接收到编码“106 108 x”，根据规则，查到 106 在编码表中，则读取一个 token，便可建立一个内容为“read”的树节点；接下来，108 不在编码表，而在解释编码表中，所以除了读入“108”外，还要继续读入一个 token “x”，由于它们之间存在上下层关系，所以需要建立一个“identifier ”->“x”

关系的子树。

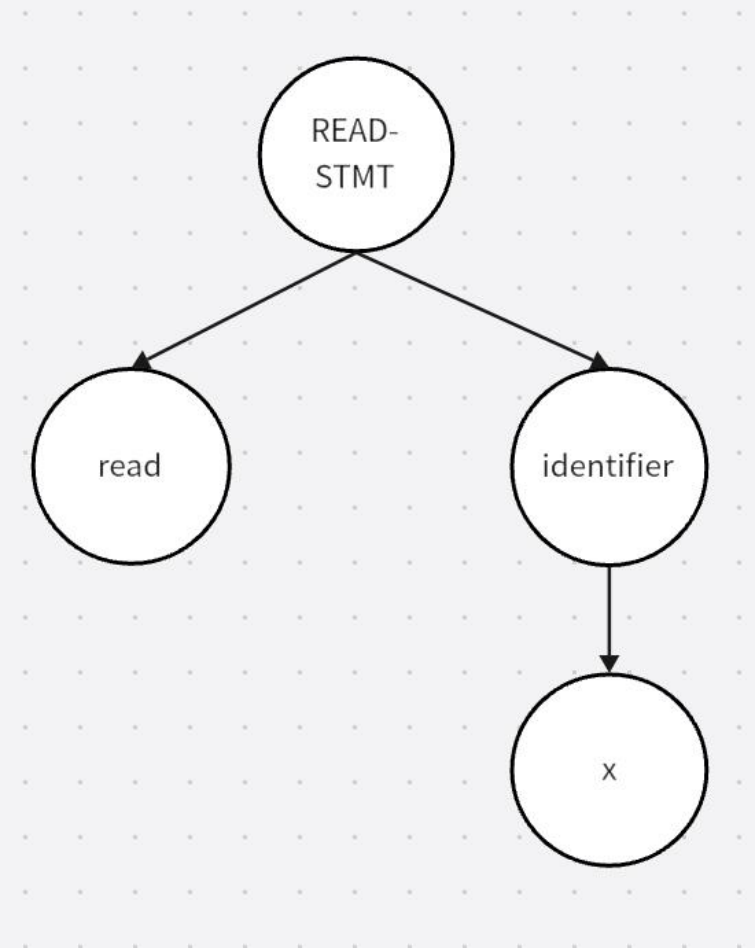

图 17　由语句构建的树

如果忽略所有解释，就会发现上述编码解码为“read identifier”，这正是SLR(1)分析时输入栈的内容。而输入栈 FIFO 的特性与队列一致，所以我们用队列管理建立的节点，建立单个节点或父子节点时，分别将单节点和父亲节点入队。

归约时分析栈总是从末端取出若干符号，符合 FILO 特性，可以用数据结构栈管理。当发生移进时，从队列首位取出一个节点插入栈顶；当发生归约时，根据被归约的状态数量，从栈中取出相应数量的节点，它们与所归约到的符号之间可以建立类似树的父子关系，例如，“READ-STMT -> read identifier”，则建立一个新节点“READ-STMT”，把它作为节点“read”和“identifier”的父亲，然后把新节点压入栈中。

特殊情况是当发生了空串的归约，则无需从栈中取出节点，而是直接创建所归约符号的节点入栈即可。

若最后达到接受状态，算法会返回语法树的根节点，根节点的内容是接受符号。

编码“106 108 x”最后会建立如下的语法树：

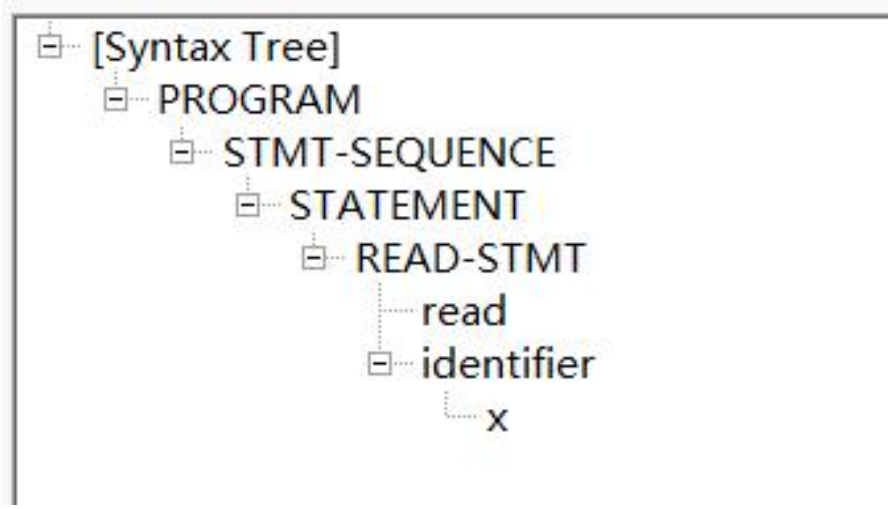

图 18　语法树

四、参考文献

Kenneth C.Louden．编译原理及实践[M]．机械工业出版社．冯博琴译．2004 年 2 月 1 日。