湖南大学

HUNAN UNIVERSITY

课程实验报告

课　程　名　称：　编译原理

实验项目名称：　正则运算表达式的 DFA 构建

专　业　班　级：　软件 2005

姓　　　　　名：　邹佳骏

学　　　　　号：　202026010501

指　导　教　师：　杨金民

完　成　时　间：　2023 年 3 月 7 日

信息科学与工程学院

实验题目：

实验一、正则运算表达式的 DFA 构建

实验目的：

将理论知识（如何构建一个最简 NFA）付诸于代码实践，巩固上课所学；

从代码层次深入理解一个编译器是如何进行词法分析；

从代码层次掌握如何将 NFA 转化为 DFA。

实验环境：

PC、Windows 操作系统、Dev-C++

实验内容及操作步骤：

实验内容：

1. 基于上述数据结构的定义，针对字符集的创建，实现如下函数：

int range (char fromChar, char toChar);　// 字符的范围运算

int union(char c1, char c2);　// 字符的并运算

int union(int charSetId, char c)；// 字符集与字符之间的并运算

int union(int charSetId1,int charSetId2)；//字符集与字符集的并运算

int difference(int charSetId, char c);　// 字符集与字符之间的差运算

这 5 个函数都会创建一个新的字符集对象，返回值为字符集 id。创建字符集，表现为往字符集表中添加新的行。当一个字符集包含多个段时，便会在字符集表中有多行，一行记录一段。

2. 基于上述 NFA 的数据结构定义，请按照最简 NFA 构造法，实现如下函数：

Graph * generateBasicNFA(DriverType driverType，int driverId);

Graph * union(Graph *pNFA1, Graph *pNFA2);　// 并运算

Graph * product(Graph *pNFA1, Graph *pNFA2); // 连接运算

Graph * plusClosure(Graph *pNFA) //正闭包运算

Graph * closure(Graph *pNFA) // 闭包运算

Graph * zeroOrOne(Graph *pNFA);　// 0 或者 1 个运算。

其中第 1 个函数 generateBasicNFA 是针对一个字符或者一个字符集，创建其 NFA。其 NFA 的基本特征是：只包含两个状态（0 状态和 1 状态），且结束状态（即 1 状态）无出边。后面 5 个函数则都是有关 NFA 的组合，分别对应 5 种正则运算，创建一个新的 NFA 作为返回值。

3. 针对上述 NFA 的数据结构定义，实现如下函数：

(1) 子集构造法中的 3 个函数：move，e_closure，DTran；

(2) 将 NFA 转化为 DFA 的函数：

　　Graph * NFA_to_DFA(Graph *pNFA);

　　在这个函数的实现代码中，会创建一个 DFA，作为返回值。

4. 实现了上述函数之后，请以正则表达式(a|b)*abb 来测试，检查实现代码的正确性。

然后再以 TINY 语言的词法来验证程序代码的正确性，得出 TINY 语言的词法的 DFA；

操作步骤：

数据结构：

```
enum OperandType { CHAR_OT, CHARSET_OT, REGULAR };
enum DriverType { NULL_DT, CHAR_DT, CHARSET_DT };
enum StateType { MATCH, UNMATCH };
enum LexemeCategory {
    INTEGER_CONST,        // 整数常量
    FLOAT_CONST,          // 实数常量
    SCIENTIFIC_CONST,     // 科学计数法常量
    NUMERIC_OPERATOR,     // 数值运算词
    NOTE,                 // 注释
    STRING_CONST,         // 字符串常量
    SPACE_CONST,          // 空格常量
    COMPARE_OPERATOR,     // 比较运算词
    ID,                   // 变量词
    LOGIC_OPERATOR        // 逻辑运算词
};
```

OperandType 表示正则表达式的操作数类型，DriverType 表示驱动字符的类型，StateType 表示状态的 type 属性，LexemeCategory 表示状态的 category 值。

```
class State {
  public:
    int stateId;
    StateType type;
    LexemeCategory category;

    State();
    State(State *state);
};
```

```
class Edge {
  public:
    int fromState;
    int nextState;
    int driverId;
    DriverType type;

    Edge();
    Edge(Edge *edge);
};
```

```
class Graph {
  public:
    int graphId;
    int numOfStates;
    list <Edge *>*pEdgeTable;
    list <State *>*pStateTable;

    Graph();
    Graph(Graph *pNFA);
};
```

```
// 正则运算式
class regularExpression {
  public:
    int regularId;
    string name;
    char operatorSymbol;    //正则运算符，共有 7 种：'=', '~', '-', '|', '.', '*', '+', '?'
    int operandId1;         //左操作数
    int operandId2;         //右操作数
    OperandType type1;      //左操作数的类型
    OperandType type2;      //右操作数的类型
    OperandType resultType; //运算结果的类型
    LexemeCategory category; // 词的 category 属性值
    Graph *pNFA;            //对应的 NFA
};
```

```
// 字符集
class CharSet {
  public:
    int indexId;     //字符集 id
    int segmentId;   //字符集中的段 id。一个字符集可以包含多个段
    char fromChar;   //段的起始字符
    char toChar;     //段的结尾字符
};
```

状态、边、图、正则运算式、字符集的数据结构。

```
int serialCharSetId = 0;
int serialSegmentId = 0;
int serialGraphId = 0;
```

设置 3 个全局变量来分别表示字符集的序号、字符集的段的序号、图的序号。

```
// 正则运算表
list<regularExpression *> *pRegularTable;
// 字符集表
list<CharSet *> *pCharSetTable;
```

创造两个 list 来分别存储正则运算式和字符集。

1.int range (char fromChar, char toChar);　// 字符的范围运算

```
// 字符的范围运算
int range(char fromChar, char toChar) {
    CharSet *charSet = new CharSet();
    charSet->indexId = ++serialCharSetId;
    charSet->segmentId = ++serialSegmentId;
    charSet->fromChar = fromChar;
    charSet->toChar = toChar;
    pCharSetTable->push_back(charSet);
    return serialCharSetId;
}
```

直接创建一个字符集，利用全局变量 serialCharSetId 和 serialSegmentId 作为字符集 id 和段 id，以 fromChar 和 toChar 作为段的头和尾。

2.int union(char c1, char c2);　// 字符的并运算

```
// 字符的并运算
int unionFunc(char c1, char c2) {
    bool includeFlag = false;

    CharSet *charSet1 = new CharSet();
    charSet1->indexId = ++serialCharSetId;
    charSet1->segmentId = ++serialSegmentId;
    if (c2 == c1 - 1) {
        charSet1->fromChar = c2;
        includeFlag = true;
    } else {
        charSet1->fromChar = c1;
    }
    if (c2 == c1 + 1) {
        charSet1->toChar = c2;
        includeFlag = true;
    } else {
        charSet1->toChar = c1;
    }
    pCharSetTable->push_back(charSet1);

    if (c1 == c2) includeFlag = true;

    if (!includeFlag) {
        CharSet *charSet2 = new CharSet();
        charSet2->indexId = serialCharSetId;
        charSet2->segmentId = ++serialSegmentId;
        charSet2->fromChar = c2;
        charSet2->toChar = c2;
        pCharSetTable->push_back(charSet2);
    }
    return serialCharSetId;
}
```

创建字符集，确定字符集的 id 和段 id，然后分情况。includeFlag 表示 c1 和 c2 是否是相邻的字符，如果 incldueFlag 是 ture，就只用创建一个字符集，一个段，否则需要创建相同字符集 id 的两个段。

3.int union(int charSetId, char c)；// 字符集与字符之间的并运算

```
// 字符集与字符之间的并运算
int unionFunc(int charSetId, char c) {
    ++serialCharSetId;
    bool includeFlag = false;
    for (list<CharSet *>::iterator it = pCharSetTable->begin(); it != pCharSetTable->end(); ++it) {
        if ((*it)->indexId == charSetId) {
            CharSet *tmpCharSet = new CharSet();
            tmpCharSet->indexId = serialCharSetId;
            tmpCharSet->segmentId = (*it)->segmentId;
            if (c == (*it)->fromChar - 1) {
                tmpCharSet->fromChar = c;
                includeFlag = true;
            } else {
                tmpCharSet->fromChar = (*it)->fromChar;
            }
            if (c == (*it)->toChar + 1) {
                tmpCharSet->toChar = c;
                includeFlag = true;
            } else {
                tmpCharSet->toChar = (*it)->toChar;
            }
            if (c >= tmpCharSet->fromChar && c <= tmpCharSet->toChar) includeFlag = true;
            pCharSetTable->push_back(tmpCharSet);
        }
    }
    if (!includeFlag) {
        CharSet *charSet = new CharSet();
        charSet->indexId = serialCharSetId;
        charSet->segmentId = ++serialSegmentId;
        charSet->fromChar = c;
        charSet->toChar = c;
        pCharSetTable->push_back(charSet);
    }
    return serialCharSetId;
}
```

includeFlag 表示 c 是否可以并入 charSetId 下某一段中，首先在字符集表中找到 id 是 charSetId 的段，分别将 c 和该段的头尾进行比较，如果是某一段头的前一个字符或者尾的后一个字符，就直接修改该段；如果在该段中，就直接跳过；如果 includeFlag 为 false，即不在任何一段中，也不与任何一段相邻，就创一个新段。

4.int union(int charSetId1,int charSetId2)；//字符集与字符集的并运算

```
//字符集与字符集的并运算
int unionFunc(int charSetId1,int charSetId2) {
    ++serialCharSetId;
    map<char, int> existMap;
    int minChar = 127;
    int maxChar = 0;
    for (list<CharSet *>::iterator it = pCharSetTable->begin(); it != pCharSetTable->end(); ++it) {
        if ((*it)->indexId == charSetId1 || (*it)->indexId == charSetId2) {
            for (char i = (*it)->fromChar; i <= (*it)->toChar; ++i) {
                if (existMap.count(i) == 0) {
                    existMap[i] = 1;
                    if (i < minChar) minChar = i;
                    if (i > maxChar) maxChar = i;
                }
            }
        }
    }
    bool beginFlag = true;
    for (int i = minChar; i <= maxChar + 1; ++i) {
        if (!beginFlag && existMap.count(i)) {
            beginFlag = true;
            minChar = i;
        }
        if (beginFlag) {
            if (existMap.count(i) == 0) {
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = ++serialSegmentId;
                tmpCharSet->fromChar = minChar;
                tmpCharSet->toChar = i - 1;
                pCharSetTable->push_back(tmpCharSet);
                beginFlag = false;
            }
        }
    }
    return serialCharSetId;
}
```

首先遍历整个字符集表，找到所有属于 charSetId1 和 charSetId2 的段，将里面所有的字符加入 existMap 的 map 中。设置一个 beginFlag 的 bool 类型，表示是否是新段的一部分。遍历所有的字符，如果是开头，并且不在 map 中，就表示这一段已经找到了，就创建新段。然后一直等到字符存在在 map 中，就相当于找到新头，就将 beginFlag 置为 true，继续遍历，直到段结束。

5.int difference(int charSetId, char c); // 字符集与字符之间的差运算

```
// 字符集与字符之间的差运算
int difference(int charSetId, char c) {
    ++serialCharSetId;
    for (list<CharSet *>::iterator it = pCharSetTable->begin(); it != pCharSetTable->end(); ++it) {
        if ((*it)->indexId == charSetId) {
            if (c == (*it)->fromChar) {
                // c为头，取余下一段
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = ++serialSegmentId;
                tmpCharSet->fromChar = (*it)->fromChar + 1;
                tmpCharSet->toChar = (*it)->toChar;
                pCharSetTable->push_back(tmpCharSet);
            } else if (c > (*it)->fromChar && c < (*it)->toChar) {
                // c 为中间，取前后两段
                CharSet *tmpCharSet1 = new CharSet();
                tmpCharSet1->indexId = serialCharSetId;
                tmpCharSet1->segmentId = ++serialSegmentId;
                tmpCharSet1->fromChar = (*it)->fromChar;
                tmpCharSet1->toChar = c - 1;
                pCharSetTable->push_back(tmpCharSet1);

                CharSet *tmpCharSet2 = new CharSet();
                tmpCharSet2->indexId = serialCharSetId;
                tmpCharSet2->segmentId = ++serialSegmentId;
                tmpCharSet2->fromChar = c + 1;
                tmpCharSet2->toChar = (*it)->toChar;
                pCharSetTable->push_back(tmpCharSet2);
            } else if (c == (*it)->toChar) {
                // c为尾，取前段
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = ++serialSegmentId;
                tmpCharSet->fromChar = (*it)->fromChar;
                tmpCharSet->toChar = (*it)->toChar - 1;
                pCharSetTable->push_back(tmpCharSet);

            } else {
                // c在范围外，不管
                CharSet *tmpCharSet = new CharSet();
                tmpCharSet->indexId = serialCharSetId;
                tmpCharSet->segmentId = (*it)->segmentId;
                tmpCharSet->fromChar = (*it)->fromChar;
                tmpCharSet->toChar = (*it)->toChar;
                pCharSetTable->push_back(tmpCharSet);
            }
        }
    }
    return serialCharSetId;
}
```

首先在字符集表中找到所有属于该字符集的段，有四种情况分别处理。如果 c 是某一段的头，就取掉头，取余下的为一段；如果 c 在某一段的中间，就分为两段，取掉中间的 c；如果 c 是某一段的尾，就取掉尾，取余下的为一段；如果 c 不属于这一段，就不处理。

6.Graph * generateBasicNFA(DriverType driverType，int driverId);

```
Graph * generateBasicNFA(DriverType driverType, int driverId) {
    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = 2;

    // 创建首尾状态
    int serialStateId = -1;
    State *state1 = new State();
    state1->stateId = ++serialStateId;
    state1->type = UNMATCH;
    State *state2 = new State();
    state2->stateId = ++serialStateId;
    state2->type = MATCH;

    // 添加状态列表
    list<State *> *stateTable = new list<State *>();
    stateTable->push_back(state1);
    stateTable->push_back(state2);
    graph->pStateTable = stateTable;

    // 创建边
    Edge *edge = new Edge();
    edge->fromState = state1->stateId;
    edge->nextState = state2->stateId;
    edge->driverId = driverId;
    edge->type = driverType;

    // 添加边列表
    list<Edge *> *edgeTable = new list<Edge *>();
    edgeTable->push_back(edge);
    graph->pEdgeTable = edgeTable;

    return graph;
}
```

创建一个只有起始状态和终止状态两个状态的图。首先新建一个 graph，设置序号和状态数，然后创建首尾状态，添加状态列表，创建连接起始状态和终止状态的边，并将边加入图的边列表。

7.Graph * union(Graph *pNFA1, Graph *pNFA2);　// 并运算

```
// 并运算
Graph * unionFunc(Graph *pNFA1, Graph *pNFA2) {
    Graph *newGraph1 = pNFA1;
    Graph *newGraph2 = pNFA2;
    // 预处理，处理成为初始无入，最终无出，可能产生垃圾
    if (graphHasIn(pNFA1)) newGraph1 = graphAddBeginState(newGraph1);
    if (graphHasOut(pNFA1)) newGraph1 = graphAddEndState(newGraph1);
    if (graphHasIn(pNFA2)) newGraph2 = graphAddBeginState(newGraph2);
    if (graphHasOut(pNFA2)) newGraph2 = graphAddEndState(newGraph2);

    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = newGraph1->numOfStates + newGraph2->numOfStates - 2;

    // 添加状态
    list<State *> *stateTable = new list<State *>();
    // 起始状态
    State *beginState = new State();
    beginState->stateId = 0;
    beginState->type = UNMATCH;
    stateTable->push_back(beginState);
```

根据最简 NFA 的创建规则，针对 pNFA1 和 pNFA2，如果起始状态有入边，就在起始状态添加一个状态，如果结束状态有出边，就在结束状态添加一个状态。创建新的图，初始化图的状态。

```
    // newGraph1状态
    for (list<State *>::iterator it = newGraph1->pStateTable->begin(); it != newGraph1->pStateTable->end(); ++it) {
        if ((*it)->stateId == 0) continue;  // 不添加第一个状态
        if ((*it)->stateId == newGraph1->numOfStates - 1) continue; // 不添加最后一个状态
        State *state = new State();
        state->stateId = (*it)->stateId;
        state->type = UNMATCH;
        state->category = (*it)->category;
        stateTable->push_back(state);
    }
    // newGraph2状态
    for (list<State *>::iterator it = newGraph2->pStateTable->begin(); it != newGraph2->pStateTable->end(); ++it) {
        if ((*it)->stateId == 0) continue;  // 不添加第一个状态
        if ((*it)->stateId == newGraph2->numOfStates - 1) continue; // 不添加最后一个状态
        State *state = new State();
        state->stateId = (*it)->stateId + newGraph1->numOfStates - 1;
        state->type = UNMATCH;
        state->category = (*it)->category;
        stateTable->push_back(state);
    }
    // 最终状态
    State *endState = new State();
    endState->stateId = newGraph1->numOfStates + newGraph2->numOfStates - 3;
    endState->type = MATCH;
    stateTable->push_back(endState);
    graph->pStateTable = stateTable;
```

对于第一个图，不添加第一个状态和最后一个状态，对于第二个图，同样不添加第一个状态和最后一个状态。

```
    // 添加边
    list<Edge *> *edgeTable = new list<Edge *>();
    // 第一个图，仅更改末尾边
    for (list<Edge *>::iterator it = newGraph1->pEdgeTable->begin(); it != newGraph1->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        tmpEdge->fromState = (*it)->fromState;
        if ((*it)->nextState == newGraph1->numOfStates - 1) {    // 最终状态入边下一状态改变
            tmpEdge->nextState = newGraph1->numOfStates + newGraph2->numOfStates - 3;
        } else {
            tmpEdge->nextState = (*it)->nextState;
        }
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    // 第二个图，更改所有边的首尾状态
    int baseStateId = newGraph1->numOfStates - 2;
    for (list<Edge *>::iterator it = newGraph2->pEdgeTable->begin(); it != newGraph2->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        if ((*it)->fromState == 0) {    // 最初状态出边，从0出，连接到id + baseId
            tmpEdge->fromState = 0;
        } else {
            tmpEdge->fromState = (*it)->fromState + baseStateId;
        }
        tmpEdge->nextState = (*it)->nextState + baseStateId;
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    graph->pEdgeTable = edgeTable;
    return graph;
}
```

对于第一个图，只改变末尾边，最终状态的下个状态和后面的状态合并。对于第二个图，改变所有边的首尾，连接到第一个图为基础的后面。

8.Graph * product(Graph *pNFA1, Graph *pNFA2); // 连接运算

```
// 连接运算
Graph * product(Graph *pNFA1, Graph *pNFA2) {
    Graph *newGraph1 = pNFA1;
    Graph *newGraph2 = pNFA2;
    if (graphHasOut(pNFA1) && graphHasIn(pNFA2)) {
        // 新增一个状态
        newGraph1 = graphAddEndState(pNFA1);
        // 将图1末尾状态当作图2开始状态
        State *endState = newGraph1->pStateTable->back();
        State *beginState = newGraph2->pStateTable->front();
        endState->type = UNMATCH;
        endState->category = beginState->category;
    }

    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = newGraph1->numOfStates + newGraph2->numOfStates - 1;

    // 添加状态
    list<State *> *stateTable = new list<State *>();
    // 添加图1所有状态
    for (list<State *>::iterator it = newGraph1->pStateTable->begin(); it != newGraph1->pStateTable->end(); ++it) {
        State *tmpState = new State();
        tmpState->stateId = (*it)->stateId;
        tmpState->type = UNMATCH;   // 图一所有状态均为unmatch
        tmpState->category = (*it)->category;
        stateTable->push_back(tmpState);
    }
    // 添加图2除初始状态外所有状态，状态ID增加
    int baseStateId = newGraph1->numOfStates - 1;
    for (list<State *>::iterator it = newGraph2->pStateTable->begin(); it != newGraph2->pStateTable->end(); ++it) {
        State *tmpState = new State();
        if ((*it)->stateId == 0) continue;
        tmpState->stateId = (*it)->stateId + baseStateId;
        tmpState->type = (*it)->type;
        tmpState->category = (*it)->category;
        stateTable->push_back(tmpState);
    }
    graph->pStateTable = stateTable;


    // 添加边
    list<Edge *> *edgeTable = new list<Edge *>();
    // 添加图1所有边
    for (list<Edge *>::iterator it = newGraph1->pEdgeTable->begin(); it != newGraph1->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        tmpEdge->fromState = (*it)->fromState;
        tmpEdge->nextState = (*it)->nextState;
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    // 添加图2所有边
    for (list<Edge *>::iterator it = newGraph2->pEdgeTable->begin(); it != newGraph2->pEdgeTable->end(); ++it) {
        Edge *tmpEdge = new Edge();
        tmpEdge->driverId = (*it)->driverId;
        tmpEdge->fromState = (*it)->fromState + baseStateId;
        tmpEdge->nextState = (*it)->nextState + baseStateId;
        tmpEdge->type = (*it)->type;
        edgeTable->push_back(tmpEdge);
    }
    graph->pEdgeTable = edgeTable;
    return graph;
}
```

如果 pNFA1 有出边且 pNFA2 有入边，根据最简 NFA 的构建规则，需要加入空变换。然后创建新图，加入 pNFA1 和 pNFA2 的状态和边

9.Graph * plusClosure(Graph *pNFA) //正闭包运算

```
//正闭包运算
Graph * plusClosure(Graph *pNFA) {
    // 增加一条边即可
    Graph *graph = new Graph(pNFA);
    Edge *edge = new Edge();
    edge->driverId = 0;
    edge->type = NULL_DT;
    edge->fromState = pNFA->numOfStates - 1;
    edge->nextState = 0;
    graph->pEdgeTable->push_back(edge);
    return graph;
}
```

只需要增加一条边

10.Graph * closure(Graph *pNFA) // 闭包运算

```
// 闭包运算
Graph * closure(Graph *pNFA) {
    Graph *graph = new Graph(pNFA);
    // 是否可以化简
    bool hasIn = graphHasIn(graph);
    bool hasOut = graphHasOut(graph);
    if (!hasIn && !hasOut && graph->numOfStates == 2) {
        // 保留唯一状态
        graph->numOfStates = 1;
        list<State *>::iterator itState = graph->pStateTable->begin();
        State *beginState = new State(*itState);
        beginState->type = MATCH;
        graph->pStateTable->clear();
        graph->pStateTable->push_back(beginState);
        // 处理边
        for (list<Edge *>::iterator it = graph->pEdgeTable->begin(); it != graph->pEdgeTable->end();) {
            if ((*it)->type != NULL_DT) {
                (*it)->nextState = 0;
                ++it;
            } else {
                graph->pEdgeTable->erase(it);
            }
        }
    } else {
        // 原末->原初，终出->终末
        // 添加返回边
        Edge *edge = new Edge();
        edge->driverId = 0;
        edge->type = NULL_DT;
        edge->fromState = pNFA->numOfStates - 1;
        edge->nextState = 0;
        graph->pEdgeTable->push_back(edge);
        // 预处理图
        if (graphHasIn(graph)) graph = graphAddBeginState(graph);
        if (graphHasOut(graph)) graph = graphAddEndState(graph);
        // 添加跳过边
        edge = new Edge();
        edge->driverId = 0;
        edge->type = NULL_DT;
        edge->fromState = 0;
        edge->nextState = pNFA->numOfStates - 1;
        graph->pEdgeTable->push_back(edge);
    }
    return graph;
}
```

首先根据特殊正则表达式的最简 NFA 构造，如果没有出边且没有入边且只有两个状态，就符合特殊情况，能够减少为一个状态，处理边即可；如果不符合上述条件，就根据最简 NFA 规则，有入边就在开始状态前添加空变换，如果有出边，就在结束状态后添加空变换。

11.Graph * zeroOrOne(Graph *pNFA);   // 0 或者 1 个运算。

```
Graph * zeroOrOne(Graph *pNFA) {
    Graph *graph = new Graph(pNFA);
    if (graphHasIn(graph)) graph = graphAddBeginState(graph);
    if (graphHasOut(graph)) graph = graphAddEndState(graph);
    // 增加从开始状态到最终状态的边
    Edge *edge = new Edge();
    edge->driverId = 0;
    edge->type = NULL_DT;
    edge->fromState = 0;
    edge->nextState = pNFA->numOfStates - 1;
    graph->pEdgeTable->push_back(edge);

    return graph;
}
```

根据最简 NFA 的构建规则，有入边就在开始状态前添加空变换，如果有出边，就在结束状态后添加空变换。

12.move

```
list<int> *move(Graph *graph, list<int> *states, int driverId) {
    list<int> *nextStates = new list<int>();
    for (list<int>::iterator itState = states->begin(); itState != states->end(); ++itState) {
        for (list<Edge *>::iterator itEdge = graph->pEdgeTable->begin(); itEdge != graph->pEdgeTable->end(); ++itEdge) {
            if (*itState == (*itEdge)->fromState && driverId == (*itEdge)->driverId) {
                // 查重
                if (find(nextStates->begin(), nextStates->end(), (*itEdge)->nextState) == nextStates->end()) {
                    nextStates->push_back((*itEdge)->nextState);
                }
            }
        }
    }
    return nextStates;
}
```

遍历状态列表中的每一个状态，在图的所有边中遍历，如果找到是该起始状态，且驱动字符符合的边，在查重后，如果没有重复，就添加到 nextStates 的列表中。

13.e_closure

```
list<int> *eClosure(Graph *graph, list<int> *states) {
    list<int> *closureStates = new list<int>();
    queue<int> qStates;
    for (list<int>::iterator it = states->begin(); it != states->end(); ++it) {
        closureStates->push_back(*it);
        qStates.push(*it);
    }
    while (!qStates.empty()) {
        int state = qStates.front();
        qStates.pop();
        for (list<Edge *>::iterator itEdge = graph->pEdgeTable->begin(); itEdge != graph->pEdgeTable->end(); ++itEdge) {
            if (state == (*itEdge)->fromState && (*itEdge)->type == NULL_DT) {
                // 查重
                if (find(closureStates->begin(), closureStates->end(), (*itEdge)->nextState) == closureStates->end()) {
                    closureStates->push_back((*itEdge)->nextState);
                    qStates.push((*itEdge)->nextState);
                }
            }
        }
    }
    return closureStates;
}
```

将已有的 states 列表中的状态加入 e_closure 运算的集合中，全部加入 qStates 的队列中之后处理，处理 qStates 中的每个状态，遍历图中的所有边，如果起始状态是该状态且驱动字符是空变换，在查重后，如果没有重复，就加入到 e_closure 的集合中，也加入到 qStates 的队列中，每处理完一个状态，就从队列中删除。

注：这里的 DTran 实际就是将 move 的后的结果，进行空闭包运算得到结果，这里不做展示，以上函数包含了这一功能。

14.将 NFA 转化为 DFA 的函数：Graph * NFA_to_DFA(Graph *pNFA)；

```
Graph * NFA_to_DFA(Graph *pNFA) {
    // 已有的状态集合列表
    int stateListId = 0;
    list<list<int> *> *existStates = new list<list<int> *>();
    map<int, list<int> *> *existStatesMap = new map<int, list<int> *>();

    // DFA表
    vector<vector<int>> DFAtable;
    vector<int> *drivers = getAllDriver(pNFA);

    list<int> *zero = new list<int>();
    zero->push_back(0);
    list<int> *stateList = eClosure(pNFA, zero);
    existStates->push_back(stateList);
    existStatesMap->insert(pair<int, list<int> *>(0, stateList));
    free(zero);

    queue<int> qStateList;
    qStateList.push(0);
    int row = 0;
    while (!qStateList.empty()) {
        int state = qStateList.front();
        qStateList.pop();

        vector<int> rowVector;
        int len = drivers->size();
        for (int i = 0; i < len; ++i) {
            list<int> *tmpStateList = move(pNFA, existStatesMap->at(row), drivers->at(i));
            if (tmpStateList->size() == 0) rowVector.push_back(-1);
            else {
                int preId = stateListId;
                int num = checkStateExisted(existStates, tmpStateList, existStatesMap, stateListId);
                if (preId + 1 == stateListId) {
                    qStateList.push(num);
                }
                rowVector.push_back(num);
            }
        }
        DFAtable.push_back(rowVector);
        ++row;
    }
```

先得到 NFA 中所有的驱动字符，stateList 用来放 0 状态的 e_closure 的集合，加入 existStates 中作为一个 DFA 状态，将 0 状态加入 qStateList 队列。处理 qStateList 队列，针对一个 e_closure 的集合，调用 move 函数，对所有驱动字符都运行一次，没有后续就不操作，有后续状态就检查是不是已经存在的状态，如果是新状态就加入队列，这样就能得到 DFA 表。

```
    // 表转换成图
    Graph *graph = new Graph();
    graph->graphId = ++serialGraphId;
    graph->numOfStates = DFAtable.size();

    // 添加状态
    list<State *> *stateTable = new list<State *>();
    for (int i = 0; i < graph->numOfStates; ++i) {
        State *tmpState = new State();
        tmpState->stateId = i;
        tmpState->type = getType(pNFA, existStatesMap, i);
        stateTable->push_back(tmpState);
    }
    graph->pStateTable = stateTable;

    // 添加边
    list<Edge *> *edgeTable = new list<Edge *>();
    for (int i = 0; i < graph->numOfStates; ++i) {
        for (int j = 0; j < drivers->size(); ++j) {
            if (DFAtable[i][j] != -1) {
                Edge *tmpEdge = new Edge();
                tmpEdge->fromState = i;
                tmpEdge->nextState = DFAtable[i][j];
                tmpEdge->driverId = drivers->at(j);
                tmpEdge->type = getDriverType(pNFA, tmpEdge->driverId);
                edgeTable->push_back(tmpEdge);
            }
        }
    }
    graph->pEdgeTable = edgeTable;
    return graph;
}
```

根据上面的表添加状态添加边，就能得到 DFA。

运行测试程序

```
range: 1 from a to a
range: 2 from b to b
CharSet List:
1 a a
2 b b

graph a: 1
graph b: 2
graph a|b: 3
graph (a|b)*: 4
graph (a|b)*a: 5
graph (a|b)*abb: 7
```

NFA 测试

```
graphId: 7
States: 4
state Lists:
Id: 0 type: 1 category: 0
Id: 1 type: 1 category: 0
Id: 2 type: 1 category: 0
Id: 3 type: 0 category: 0
Edges: 5
edge lists:
from 0 to 0 driverId: 1 type: 1
from 0 to 0 driverId: 2 type: 1
from 0 to 1 driverId: 1 type: 1
from 1 to 2 driverId: 2 type: 1
from 2 to 3 driverId: 2 type: 1
```

DFA 测试

```
DFA: 8
graphId: 8
States: 4
state Lists:
Id: 0 type: 1 category: 0
Id: 1 type: 1 category: 0
Id: 2 type: 1 category: 0
Id: 3 type: 0 category: 0
Edges: 8
edge lists:
from 0 to 1 driverId: 1 type: 1
from 0 to 0 driverId: 2 type: 1
from 1 to 1 driverId: 1 type: 1
from 1 to 2 driverId: 2 type: 1
from 2 to 1 driverId: 1 type: 1
from 2 to 3 driverId: 2 type: 1
from 3 to 1 driverId: 1 type: 1
from 3 to 0 driverId: 2 type: 1
```

由上图可见，程序成功构建出正则表达式 **(a|b)*abb** 的 NFA 图，图编号为 7，最简 NFA 图有 4

个状态，共 5 条边， 也成功将 NFA 图转化为 DFA，转化为 DFA 后，有 4 个状态，8 条边。 以上程序结果与手画 NFA，以及利用 DTran 得到 DFA 的结果相同，可以证明代码的正确性。	
实验总结	本次实验，加深了我对如何构建一个最简 NFA 图的理解，重点掌握了合并两个 NFA 图时，要对两个起始状态和终止状态的出入边进行判断，是否添加空状态。对课上讲的倒灌有了深层次的认识。