学生学号	0121710880415	实验课成绩	

武漢理工大學

学 生 实 验 报 告 书

实验课程名称	编译原理
开 课 学 院	计算机科学与技术
指导教师姓名	王云华
学 生 姓 名	颜道江
学生专业班级	软件 1704

2019 -- 2030 学年 第 一 学期

实验课程名称：编译原理

实验项目名称	词法分析			实验成绩	
实验者	颜道江	专业班级	软件 1704	组别	
同组者				实验日期	2019/11/30

二、词法分析——NFA 确定化为 DFA

1.设计题目

把 NFA 确定化为 DFA 的算法实现。在又穷自动机的理论中的定理：设 L 为一个由不确定的有穷自动机接受的集合，则存在一个接受 L 的确定的有穷自动机。本次的实验中对实现上述定理的算法——子集法进行设计实现。

2.设计目的及设计要求

构造一程序，实现：将给定的 NFA M(其状态转换矩阵及初态、终态信息保存在指定文件中)，确定化为 DFA M。

要先实现 ε-CLOSURE 函数和 Ia 函数。输出 DFA M'(其状态转换矩阵及初态、终态信息保存在指定文件中)。

3. 设计内容

3.1 ε-CLOSURE 函数设计

令 M 是一自动机，I 是 M 的状态子集，定义 ε-CLOSURE（I）如下：

（1） 若 q∈I，则 q∈ ε-CLOSURE(I)；

（2） 若 q∈I，那么从 q 出发经任意条 ε 弧到达的状态都属于 ε-CLOSURE(I)。

根据上述的定义，设计求解算法如下：

Algorithm1 ε-CLOSURE

Input：状态集 I，词法规则 f

Output: ε-CLOSURE(I)

1. 初始化 I 中所有状态 i 的求解标志 Closure_flag[i]为 False
2. While(I 中存在状态 i 的求解标志 Closure_flag[i]为 False) do
   { if ((Closure_flag[i]==False)&&存在状态 i 经过 ε 弧到达的状态)
   遍历 f 找到所有 i 经过 ε 弧到达的状态,并设求解标志为 False；
   更新当前求解标志 Closure_flag[i]==True
   }

3.2 Ia 函数设计

状态结合 I 的 α 弧转换，表示为 move(I, α)，定义为状态集合 J,其中 J 是所有那些可从 I 中某一状态经过一条 α 弧到达的状态全体。其中我们有如下的关系：

$$I_a = \varepsilon - \text{Closure}(J)$$

Algorithm2 Move(I,arc),求解状态集 I 中转态经过一条 arc 弧度到达的边

Input：状态集 I，词法规则 f，转换边 arc
Output: 求解得到的状态集 state
1. 初始化结果状态集 state
2. for i in I:
{ if (存在和 i 连接的 arc 弧)
遍历与 i 有弧相连的所有状态 new_state
if (相连的弧为 arc)
将 new_state 加入集合 state
}

3.3 状态子集算法设计

4. 输入输出设计

5. 运行结果

三、总结

四、附录

1.2 NFA 确定化为 DFA 核心代码

1. ε-CLOSURE（I）求解核心代码

```
def epsilon_closure(f, I):
    """求状态集合 I:ε-closure(I)"""
    I = set(I)
    # 设置每个状态的标志
    closure_flag = dict()
    for i in I:
        closure_flag[i] = False  # 初始状态设置为 False
    return opt_closure(f, closure_flag)


def opt_closure(f, closure_flag):
    """被 ε-closure(I)函数进行调用，进行递归求解，直到集合不再增大"""
    for i in list(closure_flag.keys()):
        if "#" in f[i].keys() and closure_flag[i] == False:
            for new_state in f[i]["#"]:
                closure_flag[new_state] = False  # 添加新的状态，并将操作标志置为 False
            # 更改当前的标志位
            closure_flag[i] = True
```

```
            # 对新加入的状态进行递归求解
            opt_closure(f, closure_flag)
    # 没有新的状态加入就返回
    return set(closure_flag.keys())
```

2. move(I, α)求解核心代码

```
def move(f, I, arc):
    """move 函数的实现，从集合 I 中的某个状态出发，经过一条 arc 弧到达的状态"""
    new_states = set()  # 将转移状态集合初始化为空集
    for i in I:
        if arc in f[i].keys():
            for new_state in f[i][arc]:
                new_states.add(new_state)
    return new_states
```

3. 子集求解算法

```
def subSet(f, E, K_0):
    """子集构造算法,传入参数为 NFA 状态装换规则，字母表和 NFA 初始状态"""
    T = {}  # 保存构造出的所有子集,键为子集的下标，值为对应的状态集合
    flags = {}  # 每个子集设置一个标志，表明是否被标记
    relations = {}  # 子集间的转换关系
    index = 0

    # 开始，令 ε-closure(K_0)，并且将其设置为未标记状态
    # 即求 T_0 并标记
    T[index] = epsilon_closure(f, K_0)
    flags[index] = False

    while True:
        # 将子集合状态族 C 初始化，其中 T_0 为子集状态族唯一成员，之后循环更新
        # C=（T_1,T_2,...）这里用列表存储下标，下标作为字典的键可进行定位
        C = list(T.keys())
        beforeSize = len(C)  # 通过子集状态族前后变化设置循环退出条件
        for i in C:
            if flags[i] == False:
                flags[i] = True  # 标记 T
                # 构造两个状态之间的转换关系
                relations[i] = {}
                for arc in E:
                    U = epsilon_closure(f, move(f, T[i], arc))

```

```
                    # 产生的 U 不在子集状态族中
                    if U not in T.values():
                        index += 1
                        T[index] = U
                        # 并将新状态的标记为设置为 False
                        flags[index] = False
                        relations[i][arc] = index
                    # 已经存在就添加转换关系
                    else:
                        # 字典中根据 value 获得 key （因为是一一对应的关系）
                        relations[i][arc] = list(T.keys())[list(T.values()).index(U)]
        # 添加新的子集并更新 C
        C = list(T.keys())
        afterSize = len(C)
        # 判断子集合构造是否结束
        if beforeSize == afterSize:  # 已经没有新的状态需要加入
            break

    return T, relations
```