Lab Report

161250174 许元武

Motivation/Aim

在第三章，我们学习了如何进行词法分析。为了巩固所学的内容，进行本次实验来自己实现一个词法分析器生成程序。

Content description

- lex文件(my.l)
- 词法分析器-生成器源代码(/src)
- 项目构建文件 (CMakeLists.txt)
- 生成的词法分析器(parser.cpp)
- 测试输入文件(input.txt)
- 输出文件(output.txt)
- 实验报告(本文件)

Ideas/Methods

注意：本次实验采用要求中的第二种方法，即编写词法分析器的生成程序来生成一个词法分析器

- 算法步骤
  - 读入.l文件中的正则表达式
  - 对这些正则表达式进行预处理
  - 将中缀的正则表达式变为后缀形式
  - 用Thompson's construction 算法将正则表达式变为NFA
  - 将多个NFA合并为一个大NFA(实现识别多个正则表达式)
  - 将这个大NFA变为DFA
  - 根据节点的出边和可接受的正则表达式(如果有的话)简化DFA
  - 根据简化后的DFA生成parser代码
- 检验步骤
  - 编译运行main.cpp，生成parser.cpp
    - 运行main.cpp时，请将cpp.template和可执行文件放在一起
  - 编译运行生成的parser.cpp
    - 运行生成的parser.cpp时，请将input.txt(字符流文件)和可执行文件放在一起
  - 查看输出文件(output.txt)

Assumptions

- .l文件中一行代表一个正则表达式(以'#'开头的行被视为注释)
- .l文件中出现的正则表达式的运算符有 |,*,()
- '.'表示匹配任意字符
- .l文件中后面出现的RE可以引用前面的RE(通过'{}')
- 使用'\'作为转义字符对保留字符进行转义
- 识别的字符集中没有空格

Related FA descriptions

- 单个正则表达式对应的NFA(见my.l中的正则表达式)
  - 例如： `id ({letter}|_)({letter}|{digit}|_)*`
- 多个NFA合并的大NFA，可识别多个正则表达式
  - 新建一个开始节点，用epsilon边连接该节点和各个NFA的开始节点，**结束状态并不合并**
- 由上述大NFA转化而来的DFA
  - 使用 子集构造+epsilon闭包

Description of important Data Structures

NFA|DFA的定义如下：

```
class FA {
 public:
     int getStartNode(){ return startNode;}
     int getEndNode(){return endNode;}
     void setStartNode(int i){startNode = i;}
     void setEndNode(int i){endNode = i;}
     void addEdge(int start,int end,char value);
     void addNode(int i);
     FA operator+(const FA& other);
     FA operator|(const FA& other);
     FA closure();
     FA(int s,int e):startNode(s),endNode(e){edges[s] = vector<edge>();edges[e] = vector<edge>();}
     FA(){}
     map<int,vector<edge>> edges;
     vector<int> findEpsilonClosure(const vector<int>& nodes);
     vector<int> subSetConstruct(const vector<int>& nodes,char edgeValue);
 private:
     int startNode;
     int endNode;
 };
 ```

 由于DFA之于NFA仅多了两个限制条件：不能有epsilon边；同一条边只能指向一个目标节点
 而我采用`map<int,vecter<edge>>`的结构来存储边，绕过了这两个条件，所以可以用`FA`来同时代表NFA和DFA
 `edge`的结构如下：

 ```
struct edge{
    char value;
    int dest;
    edge(char v,int d):value(v),dest(d){}
    bool operator<(const edge& e)const {
        if(dest == e.dest)
            return value<e.value;
        else
            return dest<e.dest;
    }
};
```

Description of core Algorithms

- 中缀表达式转后缀表达式(操作符栈)
- 将正则表达式变为NFA的Thompson's Construction算法
- 将NFA变为DFA的 子集构造+epsilon闭包法
- DFA化简：比较节点的出边情况和可接受的正则表达式(如果有的话)

Use cases on running

参见以下文件

- /src
  - my.l
- /out
  - input.txt
  - output.txt
  - parser.cpp

Problems occurred and related solutions

- 简化DFA时不能仅根据出边情况来划分等价类
  - 解决：根据出边情况和接受的RE来算一个hash值，根据这个hash值来划分等价类
- 匹配任意字符

- 在RE中用一个特殊字符来匹配任意字符，根据DFA生成代码的时候，如果边上的值是该特殊字符，就在 `case` 最后加一个 `else` ，实现匹配所有字符

Your feelings and comments

- 基础算法很重要(比如BFS)
- 优化代码的运行速度是一件困难的事
- FA是一个很厉害的模型，可以模拟解决许多问题