2022年春季学期《编译原理》大作业

第23组 组长: 李丰杰 组员: 周磊 高乐瑜

词法分析器设计文档

词法分析器整体的设计步骤是:

1. 根据提供的标准写出各种识别类型的正则表达式
2. 将正则表达式转化为NFA
3. 将各个NFA连接起来, 形成最终的NFA
4. 将NFA输入NFA确定化算法得到DFA, 然后DFA最小化.
5. 逐行读入文件, 每行按字符读入DFA, 直到找到对应终结状态或出错.

数据结构

下面是词法分析器用到的数据结构:

```
vector<string> KeyWord; // 存放文档中要求的27个关键字
vector<string> OP;      // 存放文档中要求的13个运算符
vector<string> SE;      // 存放文档中要求的3个界符

struct dfa_status{
    int no;             // 状态编号
    int term;           // 是否为终结符 0
};
vector<dfa_status> dfa_all_status; // 存放dfa的所有状态

vector<int> nfa_matrix[Num_status][Num_edge]; // 存放nfa的矩阵
// 矩阵中i行j列的元素表示 状态i经过输入j可以达到的状态集合

int dfa_martix[Num_status][Num_edge]; // 存放dfa的矩阵
// 矩阵中i行j列的元素表示 状态i经过输入j可以达到的状态

int dfa_min_martix[Num_status][Num_edge]; // 存放最小化dfa的矩阵
// 矩阵中i行j列的元素表示: 状态i经过输入j可以达到的状态
```

```
set<int> stop_status = {4, 12, 14, 24, 31, 32, 33, 35, 37, 39, 42, 44, 46, 47, 48, 49, 50, 54, 55, 56}; // nfa中的终结状态.

map<char, int> edges; // 边集合, 使用关联容器将输入元素与矩阵的列数对应起来.
```

Build NFA算法

我们的词法分析器是根据人工写出正则表达式然后转换为nfa作为输入的, 因此在项目中存在一个 `nfa.txt` 文件, 里面存放着我们生成的nfa文件. Build NFA算法就是读入nfa文件, 填充 `nfa_martix` 矩阵, 并根据不同状态间转换的字符构造 `edges` 边集合:

```
void build_nfa(std::ifstream& nfa){
    string line; // read line
    string temp1; char temp; string temp2;
    // status1, input char, status2
    int s1,s2;
    int counter_token = 0;
    while(std::getline(nfa, line)){
        stringstream input(line);
        input >> temp1; input>> temp; input >> temp2;
        // 使用stringstream来便捷地格式化数据.
        if(edges.count(temp) == 0){
            // 如果该input字符还未出现过, 加入map中
            edges.insert(std::make_pair(temp, counter_token));
            counter_token++;
        }
        // 将状态转换存入nfa矩阵
        s1 = std:: stoi(temp1); s2 = std:: stoi(temp2);
        nfa_matrix[s1-1][edges[temp]].push_back(s2);
    }
    return;
}
```

NFA to DFA算法

我们使用课上讲述的使用 ε闭包的方式将NFA转换为DFA, 基本思想是:

> “
>
> 1. 首先从S_0出发, 仅经过任意条ε条箭弧所能达到的状态组成的集合叫做初态q_0
> 2. 分别从q_0出发, 经过任意$a \in \sum$的a弧转换I_a所组成的集合作为DFA的状态, 如此继续, 直到不再有新状态产生.

因此, 我们首先要做的就是完成a弧转换的实现:

因为$I_a = \varepsilon - closure(move(I,a))$, 因此我们需要得到$move(I,a)$即所有可以从状态I的某一状态经过一条a弧而到达的状态的全体.

```
set<int> moveia(set<int> init, int input)
{
    set<int> outcome;
    // 计算当前状态通过各个输入字符的可达节点
    set<int>::iterator it;
    for(it=init.begin(); it !=init.end(); ++it){
        // 如果nfa矩阵对应元素不为空, 那就加入到最终状态集中.
        for (int j = 0; j < nfa_matrix[(*it)-1][input].size(); ++j){
            outcome.insert(nfa_matrix[(*it)-1][input][j]);
        }
    }
    return outcome;
}
```

接下来是对ε 闭包的计算, 由于该算法是硬编码到我们的词法分析器中的, 因为在nfa中我们使用 `@` 代表ε ,且在edges数据结构中, 该输入对应的数字是 `1` , 所以你会看到我们只针对nfa矩阵的下标为1的列进行操作.

```
set<int> closuer(set<int> init, int input)
{
    // 对init的每一个状态, 都寻找@可达状态
    // 生成闭包
    set<int> temp = moveia(init, input); // 先计算 move(I,a)
    set<int>::iterator it;
    for(it =temp.begin(); it!=temp.end(); ++it){
        // 针对其中的每个元素, 都寻找其按@可以到达的状态
        // 然后将这个状态插入到temp集中, 继续找@可达的状态
        // 借此实现经过任意条@边的操作, 且其为一个set, 可自动去重.
        for(int n=0; n < nfa_matrix[(*it)-1][1].size(); ++n)
            temp.insert(nfa_matrix[(*it)-1][1][n]);
    }
    return temp;
}
```

最后就是把所有组件拼在一起的操作了:

```
void nfa2dfa()
{
```

```
    set<int> init= {1};// 我们nfa的起始状态只有1
    vector< set<int> > all_states; // 辅助数据结构，用于遍历和去重.
    dfa_all_status.push_back({1,false}); // 记录状态编号和是否为终结状态
    int current = 0;
    while(true){
        // 针对当前状态，每一个可能的输入都计算一个状态
        for(int input =0; input< Num_edge; ++input){
            if (input == 1) continue; // 此时就没有@的输入了
            set<int> outcome = closuer(all_states[current], input); // 求闭包
            if(outcome.size()==0) continue;
            // 没有可达节点，那么就为空，直接忽略掉
            if(find(all_states.begin(), all_states.end(), outcome)==all_states.end())
            {
                // 如果当前生成的状态在之前没有，那么就加入，并记录下该转换记录.
                all_states.push_back(outcome);

                for(it= outcome.begin(); it!=outcome.end(); ++it){
                  // 遍历看是否为终结状态
                }
                dfa_all_status.push_back(标号);// 按顺序标号
                // dfa矩阵中记录下该转换
            }else{
                // 若存在该状态，那我们只需要记录下来该转换记录即可.
            }
        }
        current++;
        if(current == all_states.size())
            break;
    }
    Dfa_status = all_states.size(); // 记录总状态数
}
```

• DFA最小化算法

算法的核心思想就是: **把DFA的状态集k分割为不相交的子集**.

DFA M =(S,∑, δ, s0, St), 最小状态为DFA M'

步骤如下:

> “
>
> 1. 构造状态的初始划分Π:终态S_t和非终态$S-S_t$两组
> 2. 对Π施用传播性原则构造新划分Π_{new}
> 3. 如$\Pi_{new}==\Pi$, 则令$\Pi_{final}=\Pi$并继续步骤4, 否则$\Pi=\Pi_{new}$重复步骤2
> 4. 为Π_{final}中的每一组选一个代表, 这些代表构成M'的状态。若s是一代表,且$\delta(s,a)=t$, 令r是t组的代

表，则M'中有一转换$\delta'(s,a)=r$. M'的开始状态是含有s_0的那组的代表, M'的终态是含有s_t的那组的代表.

5. 去掉M'中的死状态

对Π施用传播性原则构造新划分Π_{new}步骤:

> “
>
> - 假设Π被分成m个子集$\Pi=\{S_1,S_2,\ldots,S_m\}$, 且属于不同子集的状态是可区别的，检查Π的每一个子集S_i, 看是否能够进一步划分.
> - 对于某个S_i, 令$S_i=\{s_1,s_2,\ldots,s_k\}$, 若存在数据字符a使得$I_a$不全包含在现行Π的某一子集Sj 中，则将Si 一分为二：即假定状态s_1,s_2，经过a弧分别达到状态t_1,t_2, 且t_1,t_2属于现行Π的两个不同子集，那么将S_i 分成两半，一半含有
>
> $s1:S_{i1}=\{s|s\in S_i$且s经a弧到达t_1所在子集中的某状态$\}$；另一半含有$s2:S_{i2}=S_i-S_{i1}$.
> - 由于t_1,t_2是可区别的，即存在w, 使得t_1读出w而停于终态，t_2读不出w或读出w却未停于终态。因而aw可以将s_1,s_2区分开。也就是说S_{i1}和S_{i2}中的状态时可区别的.
> - $\Pi_{new}=\{S_1,S_2,\ldots,S_{i1},S_{i2},\ldots,S_m\}$

```
void dfa_min()
{
    set<int> nt; set<int> t;
    // 非终结状态 和 终结状态
    for(int i=0; i< dfa_all_status.size(); ++i){
        // 查找所有的终结态和非终结态, 加入对应的set中
    }
    vector<set<int>> all; all.push_back(nt); all.push_back(t);
    // 所有状态集合, 并将两个初始化的集合放进去
    for(int j=0; j<Num_edge; ++j){
        // 读入每种类型的边.
        for(int i = 0; i< all.size(); ++i){
            // 对每一个集合
            if(all[i].size()<=1) continue;
            // 如果集合只有1个元素, 直接continue
            set<int>::iterator its;
            set<int> new_set; // 新的集合
            for(its=all[i].begin(); its!= all[i].end(); ++its){
                // 对集合中每个元素计算该元素到达的目的状态
                // 如果现在状态集合里面有该状态, 那么就把该状态加入到新集合里
            }
            if(new_set.size()<all[i].size()){
                // 如果原状态集中状态到达两个不同的集合
```

```
                // 那么把new_set中的元素从原状态集中删除
            }else{
                // 否则新集合就删除了, 没用, 和旧集合相同
            }
            // 添加新集合
            if(new_set.size()>0)
                all.insert(all.begin()+i+1, new_set);
        }
    }
    // 然后是对dfa矩阵进行变化
    for(int i=0; i< all.size(); ++i){
        set<int>::iterator it;
        // 对于多于一个状态的新状态
        if(all[i].size() > 1){
            it = all[i].begin();
            int temp = *it;
            // 对于每个子状态都将其转化为第一个状态
            for(; it!=all[i].end(); ++it){
                for(int m=0; m< Dfa_status; ++m){
                    for(int n=0; n< Num_edge; ++n){
                        if (all[i].count(m)!=0){
                            // 如果是该状态包含的子状态, 那么将其之间的转换变为0
                        }
                        else if(std::abs(dfa_min_martix[m][n])==*it)
                            dfa_min_martix[m][n]= dfa_min_martix[m][n] >0 ? temp :
-1*temp;
                    }
                }
            }
        }
    }
    return;
}
```

• SCAN算法

在生成DFA后, 我们就可以利用该DFA针对输入字符进行一系列的状态变换, 实现识别出输入字符或者报错:

在识别输入字符的过程中, 有以下几种特殊情况:

1. 首先判断, 如果当前已经识别出GROUP 或ORDER, 那么新输入的B和Y就不会按char字符进行处理, 而是一个特殊的边.
2. 如果当前已经读入一个 `"` 字符, 那么就开始字符串识别, **按文档要求**, 在识别字符串的过程中将所有输入(除了 `"` 与行终结符)均作为字符读入, 即可以容纳前述所有类型的字符.
3. 如果是其他合法的输入字符, 那么就去 `map<char ,int>edges` 中寻找对应的状态编码.、

4. 如果上述所有情况均不满足, 那么就认为是一个非法输入字符, 词法分析器会报错.

在本词法分析器中, 一次识别完成的标志是在读入一个字符后, 状态机的当前状态回到 `0` 状态. 此时对状态进行识别: 判断是一个合法的串还是一个ERROR信息:

1. 首先判断串是否为文档中已经定义好的关键字, 操作符, 界符
2. 如果都不是, 那么判断是否为标识符, 整形数, 浮点数, 字符串等略微不确定的类型
3. 如果上述条件均不满足, 那么就exit(-1), 输出"ERROR NOT DEFINED INPUT:串"

```
int scan(std::ifstream& text)
{
    string line;
    while(std::getline(text, line)){
        // 读取一行内容
        // 去除首空格,尾空格, 保证每行都有'\n'换行符
        int state = 1; // 起始状态
        // cout << line<<endl;
        for(int i =0; i< line.length(); ++i){
            // 读每一个字符
            buffer = line.substr(0,i);
            // 判断每一个输入
            int input = judge(buffer,state, line[i]);
            if(input == -1){
                return -1;
                // 如果输入非法, 就退出程序.
            }else{
                // 否则找到读入后的状态
                state = std::abs(dfa_martix[state-1][input]);
                if(state ==0){
                    // error 或 结束一次查找.
                    // 格式化此时准备识别的串
                    // 判断终态
                    judge_final(buffer);
                    // 回到初态, 准备识别下一个串
                    state =1 ;
                    i = -1;
                    if(line[0]==' ') line.erase(0, line.find_first_not_of(" "));
                }
            }
        }
    }
    return 0;
}
```

输出格式

正确输出格式:

- 关键字: [识别的关键字] < tab > <KW,关键字序号(文档中标记)>
- 运算符: [识别的运算符] < tab > <OP,运算符序号(文档中标记)>
- 界符: [识别的界符] < tab > <SE,界符序号(文档中标记)>
- 标识符: [识别的标识符] < tab > <IDN,识别的标识符>
- 整形数: [识别的整形数] < tab > <INT,识别的整形数>
- 浮点数: [识别的浮点数] < tab > <FLOAT,识别的浮点数>
- 字符串: [识别的字符串] < tab > <STRING,识别的字符串>

错误输出:

- 未定义输入字符: ERROR, ILLEGAL INPUT: 字符
- 未定义识别对象: ERROR NOT DEFINED INPUT:识别单词

编译步骤

您可以使用两种方式来完成词法分析器的编译与测试

方式1: 使用项目中的 `Makefile` 进行编译与测试.

```
cd ./src/lexical
make
//编译源代码, 生成可执行文件
make run
//运行我们提供的测试用例
make test
//运行验收时提供的测试用例
```

方式2: 手动输入命令进行测试

Hint: 你的所使用的c++编译器必须至少支持c++11标准, 目前使用g++和clang++编译并无问题

```
cd ./src/lexical/code
g++ -o lexical.out -std=c++11 main.cpp
./lexical.out ../test.txt > outcome.tsv
diff outcome.tsv ../standard.txt
```

语法分析器设计文档

语法分析器整体的设计步骤是:

1. 根据提供的标准写出各种识别类型的正则表达式
2. 根据初始符号构造第一个规范项目集闭包
3. 根据已有的规范项目集闭包计算新的规范项目集闭包，直达不会产生新的闭包，同时记录下状态转移，即go表
4. 根据规范项目集簇和go表构建LR（1）分析表
5. 根据LR（1）分析表对输入串进行文法检查

数据结构

下面是文法分析器用到的数据结构:

```
struct Production  //产生式
{
    string left;
    vector<string> right;
};
struct Project  //规范项目
{
    int Ps_index;
    int location;
    set<string> predict;
};
struct GoAction     //go
{
    int closure_id;
    int next_id;
    string input;
};
struct ATItem {     //分析表项
    bool b0, b1; //0,0表示accept, 1,1表示规约, 1,0表示移进, 0,1表示go
    int index;
};

vector<string> Ts;                  //终结符
vector<string> Ns;                  //非终结符
vector<Production> Ps;              //产生式
vector<set<Project>> proj_fam;      //规范集簇
vector<GoAction> go_actions;        //go表
vector<map<string, ATItem>> ATable; //LR分析表
```

```
stack<string> AStack;                //状态栈
stack<int> SStack;                   //符号栈
vector<Project> static_vp;           //计算闭包时的临时储存buffer
```

求FIRST集

计算串 X1X2 … Xn的FIRST集合

步骤如下:

> 1. 向FIRST(X1X2… Xn) 加入FIRST(X1)中所有的非ε符号
> 2. 如果ε在FIRST(X1) 中，再加入FIRST(X2)中的所有非ε符号；如果ε在 FIRST(X1) 和 FIRST(X2)中，再加入FIRST(X3)中的所有非ε符号，以此类推…
> 3. 最后，如果对所有的X，ε都在FIRST(X)中，那么将ε加入到 FIRST(X1X2… Xn)中

```
伪代码
set<string> Grammar:: calculateFirstSet(vector<string>& objective)
{
  for (i ; i < objective.size(); i++)
  {
      计算当前符号的first集合（递归）
    if 有$
                    继续
    else
        跳出循环
  }
        if 所有的符号的first集合都有$，添加$
  return first_set;
}
```

CLOSURE运算，构造项目集规范族

假定I是文法G'的任一项目集，构造I的闭包 CLOSURE(I)的方法是：

> - I的任何项目都属于CLOSURE(I)；
> - 若[A→α•Bβ, a]属于CLOSURE(I)， B→η是一个产生式,那么，对FIRST(βa)中的每个终结符b, 将[B→ • η,b]加入CLOSURE(I)；
> - 重复上述两步骤直至CLOSURE(I)不再增大为止

```
伪代码
// 获取下一个closure
void Grammar:: getClosure(int index);
//根据输入符号和对应的项目进行闭包运算
set<Project> Closure(string next_s, vector<Project> vp);
//根据项目，扩充当前项目集
void expandClosure(Project temp_project);
//判断输入的规范集是否存在，不存在就添加到规范集簇中，都要返回规范集编号
int getNextClosure(set<Project> temp_closure);
//详情见代码注释
```

构造LR（1）分析表

> “
>
> - 构造拓广文法G’的LR(1)项目集规范族C={I0 ,I1 ,…,In}
> - 从Ik构造语法分析器的状态k，状态k的分析动作如下：
> - 如果[A→α·aβ, b]在Ik中，且GO(Ik , a)= Ij，则置action[k, a]为sj ，即“移动(j, a)进栈”，这里要求a必须是终结符
> - 如果[A→α·, a]在Ik中，则置action[k, a]为rj ，即按照rj归约，其中j是产生式A→α的序号
> - 如果[S’→S·, #]在Ik中，则置action[k, #]为acc，表示接受
> - 状态k的转移按照下面的方法确定：如果GO(Ik , A)= Ij,那么goto[k, A]=j
> - 其余表项设为出错
> - 初始状态是包含[S’→·S, #]的项目集构造出的状态。

```
//构造分析表
void Grammar::getAnalyzeTable() {
  ATItem temp_item;
  map<string, ATItem> temp_row;
  int next_go;
  for (int i = 0; i < proj_fam.size(); i++) {
    for (auto j = proj_fam[i].begin(); j != proj_fam[i].end(); j++) {
      if (j->Ps_index == 0 && j->location == 1) {
        temp_item.b0 = 0;
        temp_item.b1 = 0;
        temp_item.index = 0; //acc
        temp_row["#"] = temp_item;
      }
```

```
      else if (Ps[j->Ps_index].right.size() == j->location || Ps[j->Ps_index].right[0]
== "$") {
        for (auto k = j->predict.begin(); k != j->predict.end(); k++) {
          temp_item.b0 = 1;
          temp_item.b1 = 1;
          temp_item.index = j->Ps_index; //规约
          temp_row[*k] = temp_item;

        }
      }
      else {

        for (int k = 0; k < go_actions.size();k++) {
          if (go_actions[k].closure_id == i && go_actions[k].input == Ps[j-
>Ps_index].right[j->location]) {
            next_go = go_actions[k].next_id;
          }
        }
        if (isT(Ps[j->Ps_index].right[j->location])) {
          temp_item.b0 = 1;
          temp_item.b1 = 0;
          temp_item.index = next_go; //移进
          temp_row[Ps[j->Ps_index].right[j->location]] = temp_item;
        }
        else {
          temp_item.b0 = 0;
          temp_item.b1 = 1;
          temp_item.index = next_go; //go
          temp_row[Ps[j->Ps_index].right[j->location]] = temp_item;
        }
      }
    }

    ATable.push_back(temp_row);
    temp_row.clear();
  }
}
```

- **根据LR（1）分析表，移进规约输入串**

分析栈中的串和等待输入的符号串构成如下形式的三元组: (S0S1S2…Sm, #X1X2…Xm, aiai+1 …an#)

> “
>
> 1.其初态为: (S0 , #, a1a2 …aiai+1…an#)
> 2.假定当前分析栈的栈顶为状态Sm,下一个输入符号为ai ,分析器的下一个动作:

- 如果action[Sm, ai]=移进且S=GOTO[Sm,ai],则分析器执行移进, 三元组变成 (S0S1S2…SmS, #X1X2…Xmai , ai+1…an#)即分析器将输入符号ai和状态S移进栈, ai+1变成下一个符号；
- 如果action[Sm, ai]=归约A→β,则分析器执行归约, 三元组变成 (S0S1S2…Sm-rS, #X1X2…Xm-rA, aiai+1 … an#), 此处S= GOTO[Sm-r , A], r为β的长度且β=Xm-r+1Xm-r+2…Xm；
- 若action[Sm,ai]= acc,则接收输入符号串,语法分析完成；
- 若action[Sm,ai]= err,则发现语法错误,调用错误恢复（直接返回）子程序进行处理。

```
//文法分析
void Grammar::grammarCheck(vector<string> inToken, string filename) {
  SStack.push(0);
  AStack.push("#");
  inToken.push_back("#");
  ofstream file;
  file.open(filename);
  if (!file) cout << "fail!打开结果文件失败" << endl;

  int nowState = 0;
  int inIndex = 0;
  int outSeq = 1;
  string nowSymbol;
  map<string, ATItem>::iterator it_item;
  ATItem temp_item;

  while (inIndex < inToken.size()) {
    /*if (outSeq == 5)
      cout << "this is wrong" << endl;*/
    nowState = SStack.top();
    nowSymbol = inToken[inIndex];
    it_item = ATable[nowState].find(nowSymbol); //查找分析表，不存在即出错
    if (it_item == ATable[nowState].end()) {
      cout << "Wrong!!!" << endl;
      return;
    }
    temp_item = it_item->second;

    if (temp_item.b0 && temp_item.b1) {//规约
      if (Ps[temp_item.index].right[0] != "$") {
        int rl = Ps[temp_item.index].right.size();
        for (int i = 0; i < rl; i++) {
          SStack.pop();
          //检查一下ATable有问题没
          if (Ps[temp_item.index].right[rl - i - 1] == AStack.top()) AStack.pop();
          else {
```

```
            cout << "ATable is wrong!!!" << endl;
            return;
          }
        }
      }
      AStack.push(Ps[temp_item.index].left);
      SStack.push(ATable[SStack.top()][Ps[temp_item.index].left].index);
      file << outSeq << " " << temp_item.index << " " << AStack.top() << "#" <<
nowSymbol << "  reduction" << endl;
    }
    else if (temp_item.b0 && !temp_item.b1) { //移进
      SStack.push(temp_item.index);
      AStack.push(nowSymbol);
      inIndex++;
      file << outSeq << " / " << AStack.top() << "#" << nowSymbol << "  move" << endl;
    }
    else if (temp_item.b0 && !temp_item.b1) {
      cout << "wrong!!" << endl;
      return;
    }
    else { //
      file << outSeq << " / " <<"#"<<"  " << "accept" << endl;
      return;
    }

    outSeq++;
  }
}
```

LR分析表

由于表过于庞大，详情请见ATable.tsv文件

输出格式

正确输出格式:

- 全部[序号] [TAB] [选用规则序号或/] [TAB] [栈顶符号]#[面临输入符号] [TAB] [执行动作]
- accept

错误输出:

- 部分[序号] [TAB] [选用规则序号] [TAB] [栈顶符号]#[面临输入符号] [TAB] [执行动作]
- Wrong!!

编译步骤

- 直接利用VS自动编译功能，编译并运行代码