设计文档

运行方式

下面的第一步和第二步其实已经帮忙做好了。

1. 进入源代码的前端目录中，在其中的 `vue` 项目中运行下面的命令打包前端程序。

```
npm install
npm run build
```

2. 然后将打包后的 `dist` 目录中的内容放到 `electron` 项目的根目录中。需要注意的是，这里需要修改 `index.html` 中静态文件的路径，将前面的 `/` 去掉，也就是改为相对路径。

```
# 原始的
<script type="module" crossorigin src="/assets/index-b79980b0.js"></script>
<link rel="stylesheet" href="/assets/index-53c77a11.css">
# 改为下面的形式
<script type="module" crossorigin src="assets/index-b79980b0.js"></script>
<link rel="stylesheet" href="assets/index-53c77a11.css">
```

3. 运行下面的命令打包 `electron` 程序。然后就会得到一个单独的 `exe` 程序。

```
npm install
electron-builder
```

设计思路

这个实验有四个任务：

1. 计算文法的 `first` 集合。
2. 计算文法的 `follow` 集合。
3. 计算文法对应的 `LR(0) DFA` 图。
4. 判断文法是否是 `SLR(1)` 文件，如果是给出 `SLR(1)` 分析表，如果不是给出原因。

First集合的求法

首先看 `first` 集合的定义：
$FIRST(\alpha)$：可以从 α 推导得到的串的首符号的集合。其中 α 是任意的文法符，$FIRST$ 集合中的符号是终结符号。

求法：

$First$ 集合最终是对产生式右部的字符串而言的，但其关键是求出非终结符的 $First$ 集合，由于终结符的$First$集合就是它自己，所以求出非终结符的 $First$ 集合后，就可很直观地得到每个字符串的 $First$ 集合。
1. 直接收取：对形如 $U->a...$ 的产生式（其中 a 是终结符），把 a 收入到 $First(U)$ 中。
2. 反复传送：对形入 $U->P...$ 的产生式（其中 P 是非终结符），应把 $First(P)$ 中的全部内容传送到 $First(U)$ 中。

这里需要注意的第二点是，如果对于 P，存在文法规则 $P->\sigma$，那么还要将 P 相邻右边字符的 $First$ 集中的全部内容传送到 $First(U)$。

> 这里不考虑 $A->B$，$B->A$ 这种特殊情况。

关键代码如下：

```
if (nowFirstSet.count('@') > 0) {
        // 这里移除空字符是不影响原firstSet的
        nowFirstSet.erase('@');
        for (Production production: productions) {
                if (production.leftPart == it2.first) {
                        for (int i = 0; i < production.rightPart.length(); i++) {
                                char ch = production.rightPart[i];
                                // 如果当前字符处理好了，并且其中含有空字符，就把下一个放进来
                                if (solvedSet.count(ch) > 0 && ret.find(ch)->second.count('@')
> 0) {
                                        if (i < production.rightPart.length() - 1) {
                                                it2.second.insert(production.rightPart[i + 1]);
                                        } else {
                                                // 如果到了最末尾，就加入空字符进来
                                                it2.second.insert('@');
                                        }
                                } else {
                                        break;
                                }
                        }
                }
        }
}
```

Follow集合的求法

定义：

这里根据定义就很容易知道求法了，需要注意的是：如果 σ 在 $First(nextChar)$ 中，则还需要递归的加入并判断下一个 $nextChar$ 的 $nextChar$ 的 $First$ 集，直到到末尾如果当前字符的 $First$ 集中存在 σ，就把 $Follow(leftPart)$ 加入。

关键代码如下：

```
// 如果下一个相邻的字符是终结符就直接加入
if (!isNonTerminator(nextCh)) {
        ret.find(ch)->second.insert(nextCh);
}
// 如果下一个相邻的字符是非终结符就将First(nextCh)加入，
// 并且如果@在First(nextCh)中，则还需要递归的加入并判断下一个nextCh的nextCh的First集，直到到末尾如果当前字符的First集中存在@，就把Follow(leftPart)加入
else {
        // 获取下一个元素的first集
        set<char> s = firstSetMap.find(nextCh)->second;
        // 如果可能为空则还需要继续判断后面的字符，因为要注意A->BCD，其中C的first集中含有@，因此B的follow集还需要加上D的first集
        if (s.count('@') > 0) {
                // 当前已经是最后一个元素了，如果存在@，就需要加上左边的follow集
                if (i + 2 == production.rightPart.length()) {
                        ret.find(ch)->second.insert(production.leftPart);
                }
                // 如果不是最后一个元素
                for (int j = i + 2; j < production.rightPart.length(); ++j) {
```

```
                        // 一直获取并加入下个字符的first集，还是注意要去掉@，不过要先判断有没有@
                        set<char> tmpChFirstSet = firstSetMap.find(production.rightPart[j])-
>second;
                        if (tmpChFirstSet.count('@') > 0) {
                                tmpChFirstSet.erase('@');
                                // 如果是最后一个元素，则可以直接将左边的符号的follow集加入到当前ret元
素的follow集了
                                if (j == production.rightPart.length() - 1) {
                                        ret.find(ch)->second.insert(production.leftPart);
                                }
                        }
                        ret.find(ch)->second.insert(tmpChFirstSet.begin(),
tmpChFirstSet.end());
                }
        }
        // 加入的时候需要注意去掉@空字符
        s.erase('@');
        ret.find(ch)->second.insert(s.begin(), s.end());
}
```

DFA图的绘制

按照原来学过的思路，我们求 `DFA` 图之前先需要算出 `NFA` 图，然后再通过 **子集构造法** 将 `NFA` 图转换为 `DFA` 图。但是由于这里同一个状态的项目集合之间存在某种关系，因此其实我们可以直接求出文法的 `DFA` 图。

原理是我们首先计算得到起始状态。例如下面的文法：

```
E->E+n
E->n
```

起始状态为：

$\{E`->E, E->.n, E->.E+n\}$

然后我们从起始状态开始 `bfs` **深度优先遍历** ，将可能的状态转移都计算出来，然后需要注意的是，如果转移到的下一个状态的点号后面接的是非终结符，我们还需要加入以那个非终结符开头的文法到那个下一个状态。

核心代码如下：

```
dfaNodeList.push_back(firstNode);
// 由起始状态延申至其它状态
workQueue.push(0);
while (!workQueue.empty()) {
        int pNodeIdx = workQueue.front();
        workQueue.pop();
        DfaNode pNode = dfaNodeList[pNodeIdx];
        // 记录当前状态集接收到某个字符会到哪个状态，避免当前状态集前后有多个可以接收相同字符的情况
        // 例如：A->.ab，A->.ac
        map<char, int> stateMap;
        // 暂存区
```

```
        vector<DfaNode> tmpDfaNodeList;
        // 记录到达tmpDfaNodeList中某个Node经历的是哪个char字符
        map<int, char> edgeMap;
        for (Project project: pNode.projectSet) {
                DfaNode node;
                string rightPart = project.rightPart;
                int dotIdx = rightPart.find('.');
                if (dotIdx == rightPart.length() - 1) {
                        continue;
                }
                // 点号后面相邻的字符
                char dotNextChar = rightPart[dotIdx + 1];
                // 点号后移一位
                rightPart.erase(dotIdx, 1);
                rightPart.insert(dotIdx + 1, ".");

                node.projectSet.insert(Project{project.leftPart, rightPart});
                if (dotIdx == rightPart.length() - 2) {
                        // 如果以前记录了这个字符的移进
                        if (stateMap.count(dotNextChar) > 0) {
                                tmpDfaNodeList[stateMap.find(dotNextChar)-
>second].projectSet.insert(node.projectSet.begin(),

node.projectSet.end());
                        } else {
                                tmpDfaNodeList.push_back(node);
                                stateMap.insert(make_pair(dotNextChar, tmpDfaNodeList.size() -
1));
                                // 记录边
                                edgeMap.insert(make_pair(tmpDfaNodeList.size() - 1,
dotNextChar));
                        }
                        continue;
                }
                char dotNextNextChar = rightPart[dotIdx + 2];
                // 如果dotNextChar后面的字符是非终结符，还需要加上这个非终结符的产生式
                if (isNonTerminator(dotNextNextChar)) {
                        for (Production production: productions) {
                                if (production.leftPart == dotNextNextChar) {
                                        if (production.rightPart == "@") {

node.projectSet.insert(Project{production.leftPart, "."});
                                        } else {

node.projectSet.insert(Project{production.leftPart, "." + production.rightPart});
                                        }
                                }
                        }
                }

                // 如果以前记录了这个字符的移进
                if (stateMap.count(dotNextChar) > 0) {
                        tmpDfaNodeList[stateMap.find(dotNextChar)-
```

```
>second].projectSet.insert(node.projectSet.begin(),

node.projectSet.end());
                } else {
                        tmpDfaNodeList.push_back(node);
                        stateMap.insert(make_pair(dotNextChar, tmpDfaNodeList.size() - 1));
                        // 记录边
                        edgeMap.insert(make_pair(tmpDfaNodeList.size() - 1, dotNextChar));
                }
        }
        // 全部计算完后再将暂存区中的Node放入全局变量dfaNodeList中。
        // 还需要注意这里可能新产生的node会和原来的重复，我们需要判断一下有没有和原来的重复，如过重复我们
就不需要插入，而且连边就连原来的就可以了
        // 而且如果重复了，说明原来的node肯定已经遍历过了，这样我们就不需要在把这个node入queue了，不然还
是会死循环。
        for (int i = 0; i < tmpDfaNodeList.size(); ++i) {
                DfaNode node1 = tmpDfaNodeList[i];
                bool isNew = true;
                for (int j = 0; j < dfaNodeList.size(); ++j) {
                        DfaNode node2 = dfaNodeList[j];
                        if (node1 == node2) {
                                isNew = false;
                                // 关于node1的连边就要指向node2
                                dfaEdgeList.push_back(DfaEdge{pNodeIdx, j, edgeMap.find(i)-
>second});

                                break;
                        }
                }
                if (isNew) {
                        dfaNodeList.push_back(node1);
                        int newIdx = dfaNodeList.size() - 1;
                        dfaEdgeList.push_back(DfaEdge{pNodeIdx, newIdx, edgeMap.find(i)-
>second});
                        workQueue.push(newIdx);
                }
        }
}
```

SLR(1)文法的判断和分析表的计算

$SLR(1)$ 文法的判断方式:
对于任何状态 s，满足:

1. 对于在 s 中的任何项目 $A->a.Xb$，当 X 是一个终结符，且 X 在 $Follow(B)$ 中时，s 中没有完整的项目 $B->y.$。

> 对于第一点最好反证： 如果 s 中存在 $A->y.$，则 s 中其它形如 $B->a.Xb$，X 是一个终结符的项目，$Follow(A)$ 中的字符不可能存在 X。

2. 对于在 s 中的任何两个完整项目 $A->a.$ 和 $B->b.$，$Follow(A)$ 和 $Follow(B)$ 的交集为空。

算法原理：

简单LR(1)分析，或SLR(1)分析，也如上一节中一样使用了 LR(0)项目集合的DFA。但是，通过使用输入串中下一个记号来指导它的动作，它大大地提高了 LR(0)分析的能力。它通过两种方法做到这一点。首先，它在一个移进之前先考虑输入记号以确保存在着一个恰当的 DFA。其次，它使用如4.3节所构造的非终结符的 Follow集合来决定是否应执行一个归约。令人吃惊的是，先行的这个简单应用的能力强大得足以分析几乎所有的一般的语言构造。

定义：SLR(1)分析算法(SLR(1) parsing algorithm)。令s 为当前状态(位于分析栈的顶部)。则动作可定义如下：

1. 若状态s 包含了格式$A\rightarrow\alpha.X\beta$的任意项目，其中$X$是一个终结符，且$X$是输入串中的下一个记号，则动作将当前的输入记号移进到栈中，且被压入到栈中的新状态是包含了项目$A\rightarrow\alpha X.\beta$的状态。
2. 若状态s 包含了完整项目$A\rightarrow\gamma.$，则输入串中的下一个记号是在Follow(A)中，所以动作是用规则$A\rightarrow\gamma$ 归约。用规则$S'\rightarrow S$归约与接受等价，其中S是开始状态；只有当下一个输入记号是$时，这才会发生⊖。在所有的其他情况中，新状态都是如下计算的：删除串 α和所有它的来自分析栈中的对应状态。相对应地，DFA回到α开始构造的状态。通过构造，这个状态必须包括格式 $B\rightarrow\gamma.\ A\beta$的一个项目。将$A$压入到栈中，并将包含了项目$B\rightarrow\alpha A.\beta$的状态压入。
3. 若下一个输入记号都不是上面两种情况所提到的，则声明一个错误。

若上述的SLR(1)分析规则并不导致二义性，则文法为**SLR(1)**文法(SLR(1) grammar)。特别地，当且仅当对于任何状态s，以下的两个条件：

1) 对于在s 中的任何项目$A\rightarrow\alpha.X\beta$，当$X$是一个终结符，且$X$在Follow ($B$) 中时，$s$ 中没有完整的项目$B\rightarrow\gamma.$。

2) 对于在s 中的任何两个完整项目$A\rightarrow\alpha.$和$B\rightarrow\beta.$，Follow(A) ∩ Follow(B)为空。

⊖ 实际上，任何文法扩充的开始状态S' 的Follow集合总是只由$组成，这是因为$S'$ 只出现在文法规则$S'\rightarrow S$中。

```
void getSlr1Table() {
    vector<map<char, string>> ret;
    for (int i = 0; i < dfaNodeList.size(); ++i) {
        DfaNode node = dfaNodeList[i];
        map<char, string> row;
        // 通过边寻找移进
        for (DfaEdge edge: dfaEdgeList) {
            if (edge.from == i) {
                if (isNonTerminator(edge.val)) {
                    row.insert(make_pair(edge.val, to_string(edge.to)));
                } else {
                    row.insert(make_pair(edge.val, "s" + to_string(edge.to)));
                }
            }
        }
        // 通过状态里的项目寻找归约
        for (Project p: node.projectSet) {
            if (p.rightPart.find('.') == p.rightPart.length() - 1) {
```

```
                if (p.leftPart == EXT_START) {
                    row.insert(make_pair('@', "acc"));
                    continue;
                }
                set<char> followSet = followSetMap.find(p.leftPart)->second;
                for (char ch: followSet) {
                    // 去掉末尾的.，但是如果只有一个点就转为@
                    if (p.rightPart == ".") {
                        row.insert(make_pair(ch, "r(" + char2str(p.leftPart) + "->@)"));
                    } else {
                        row.insert(make_pair(ch, "r(" + char2str(p.leftPart) + "->" +
                                                 p.rightPart.substr(0, p.rightPart.length() -
1) + ")"));
                    }
                }
            }
        }
        for (char ch: charSet) {
            // 没有记录过，也就是无法移进的，也无法归约的，填入error标志
            if (row.count(ch) == 0) {
                row.insert(make_pair(ch, ""));
            }
        }
        ret.push_back(row);
    }
    slr1Table = ret;
}
```