语法分析程序的设计与实现——LR 分析方法

任飞 2021210724

目录

1 实验题目

1.1 内容

编写 LR 语法分析程序，实现对算术表达式的语法分析。要求所分析算数表达式由如下的文法产生。

$$E \to E + T|E - T|T$$
$$T \to T * F|T/F|F$$
$$F \to (E)|\text{num}$$

1.2 要求

在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

实现方法要求：

1. 构造识别该文法所有活前缀的 DFA。
2. 构造该文法的 LR 分析表。
3. 编程实现算法 4.3，构造 LR 分析程序。

2 程序设计说明

2.1 使用方法

尽管题目不要求编程实现构造识别该文法所有活前缀的 DFA 以及构造该文法的 LR 分析表，但在此还是实现了用程序生成 DFA （即项目集规范族）和 LR 分析表的功能。

程序的使用方法和 LL 分析方法的版本类似。首先在代码中用如下格式配置要解析的文法产生式（默认首字符为大写字母的是非终结符）。

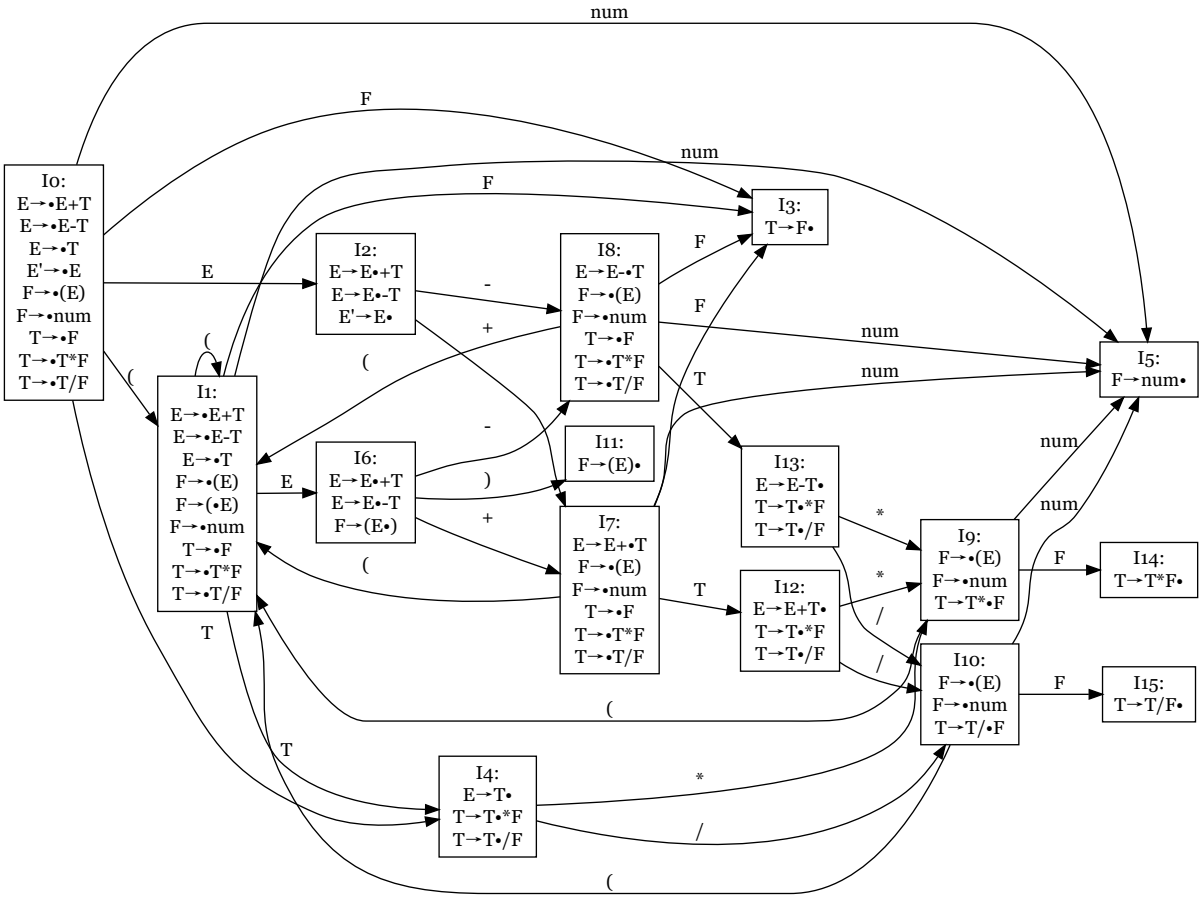
```
Grammar grammar;
grammar.addRule("E", {"E", "+", "T"});
grammar.addRule("E", {"E", "-", "T"});
grammar.addRule("E", {"T"});
grammar.addRule("T", {"T", "*", "F"});
grammar.addRule("T", {"T", "/", "F"});
grammar.addRule("T", {"F"});
grammar.addRule("F", {"(", "E", ")"});
grammar.addRule("F", {"num"});
grammar.setStart("E");
```

然后程序会输出原始产生式以及拓广文法产生式。由于题目要求的文法是 SLR(1) 文法（见下文），程序只支持自动生成 SLR(1) 分析表。如果文法不是 SLR(1) 文法，程序会输出 `Not SLR(1) grammar!` 并退出。如果文法是 SLR(1) 文法，程序会输出使用 Graphviz 格式表示的 DFA 以及构造的 SLR(1) 分析表。以题目要求的文法为例，会输出如下内容：

```
E → E+T|E-T|T
F → (E)|num
T → T*F|T/F|F

E → E+T|E-T|T
E' → E
F → (E)|num
T → T*F|T/F|F

（省略 Graphviz 格式表示的 DFA）

                  action                                  goto
      $     (     )     *     +     -     /     num   E     F     T
0           S1                                  S5    2     3     4
1           S1                                  S5    6     3     4
2     ACC                     S7    S8
3     R8          R8    R8    R8    R8    R8
4     R2          R2    S9    R2    R2    S10
5     R5          R5    R5    R5    R5    R5
6                 S11         S7    S8
```

7		S1						S5	3	12
8		S1						S5	3	13
9		S1						S5	14	
10		S1						S5	15	
11	R4		R4	R4	R4	R4	R4			
12	R0		R0	S9	R0	R0	S10			
13	R1		R1	S9	R1	R1	S10			
14	R6		R6	R6	R6	R6	R6			
15	R7		R7	R7	R7	R7	R7			

DFA 图形如下



从图中可以发现该文法存在潜在的移进-归约冲突，不是 LR(0) 文法，但计算 FOLLOW 集后可以发现该文法是 SLR(1) 文法。

接下来等待用户输入要解析的字符串，输入后程序会依次输出每一步的分析过程（即规范规约），如果输入的字符串符合文法，最后会输出 `Accept!`，否则会输出 `Reject!`。

程序内置了一个简单词法分析器，可以将输入的非负整数转换为 `num`，将输入的 `+`、`-`、`*`、`/`、`(`、`)` 转换为对应的符号，遇到 `$` 或 `EOF` 表示输入结束，忽略空白字符，遇到其他字符则报错。

2.2 程序结构

程序结构基本继承自 LL 分析方法版本的程序，由 `Grammar` 和 `LR1Table` 这两个主要的类和多种辅助数据结构类型构成。

语法分析中的符号（包括终结符和非终结符）使用 `Symbol` 类型表示，`vector<Symbol>` 被定义为 `Phrase`，即短语。

`map<Symbol, vector<Phrase>>` 类型表示产生式集合，加上初始符号就构成了一个文法，即 `Grammar` 类。`Grammar` 类提供了求 FIRST 集、求 FOLLOW 集、构造 SLR(1) 分析表的接口。

`map<pair<int, Symbol>, Action>` 和 `map<pair<Symbol, Symbol>, Phrase>` 类型分别表示 LR(1) 分析表的 action 表和 goto 表。尽管本程序实际构造的是 SLR(1) 分析表，但 LR(1) 分析表和 SLR(1) 分析表在结构和用法上完全一致，因此直接使用 `LR1Table` 命名。`LR1Table` 类提供了使用该表对输入进行分析的接口。接口采用“语法分析驱动”的模式，传入一个返回 token 的回调函数，每当语法分析需要一个 token（即终结符号）时调用一次该函数。

2.3 算法实现

求 FIRST 集

`getFirstSet` 函数可以求出指定短语的 FIRST 集，实现方法为：如果是第一个符号终结符则直接加入 FIRST 集；如果第一个符号是非终结符则将其 FIRST 集（除了 ε ）加入 FIRST 集，如果该符号的 FIRST 集中包含 ε ，则继续处理下一个符号，直到遇到终结符或者某个符号的 FIRST 集不包含 ε 。如果所有符号的 FIRST 集都包含 ε ，则将 ε 加入 FIRST 集。

```
1  set<Symbol> Grammar::getFirstSet(const Phrase &phrase) const {
2      auto iter = firstSetCache_.find(phrase);
3      if (iter != firstSetCache_.end()) {
4          return iter->second;
5      }
6      firstSetCache_[phrase] = {}; // 避免无穷递归
7      set<Symbol> firstSet;
8      bool allHaveEpsilon = true;
9      for (const auto &symbol : phrase) {
10         if (symbol.isTerminal()) {
11             firstSet.insert(symbol);
12             allHaveEpsilon = false;
13             break;
14         } else {
15             bool haveEpsilon = false;
16             for (const auto &right : rules_.at(symbol)) {
17                 auto s = getFirstSet(right);
18                 if (s.find(Symbol::epsilon) != s.end()) {
19                     s.erase(Symbol::epsilon);
20                     haveEpsilon = true;
21                 }
22                 firstSet.merge(s);
23             }
24             if (!haveEpsilon) {
25                 allHaveEpsilon = false;
```

```
                break;
            }
        }
    }
    if (allHaveEpsilon) {
        firstSet.insert(Symbol::epsilon);
    }
    firstSetCache_[phrase] = firstSet;
    return firstSet;
}
```

求 FOLLOW 集

与求 FIRST 集不同，`getAllFollowSets` 函数一次性求出所有非终结符的 FOLLOW 集。由于非迭代实现方法较为复杂，这里直接采用教材上的定义 4.4 迭代求解。

```
map<Symbol, set<Symbol>> Grammar::getAllFollowSets() const {
    map<Symbol, set<Symbol>> followSets;
    followSets[start_].insert(Symbol::end);
    bool changed = false;
    do {
        changed = false;
        for (const auto &[left, rightList] : rules_) {
            for (const auto &right : rightList) {
                int len = right.size();
                for (int i = 0; i < len; i++) {
                    if (right[i].isTerminal()) {
                        continue;
                    }
                    auto oldFollowSet = followSets[right[i]];
                    Phrase beta(right.begin() + i + 1, right.end());
                    auto betaFirstSet = getFirstSet(beta);
                    if (betaFirstSet.find(Symbol::epsilon) != betaFirstSet.end())
    {
                        betaFirstSet.erase(Symbol::epsilon);
                        followSets[right[i]].insert(followSets[left].begin(),
                                                    followSets[left].end());
                    }
                    followSets[right[i]].merge(betaFirstSet);
                    if (oldFollowSet != followSets[right[i]]) {
                        changed = true;
                    }
                }
```

```
            }
        }
    } while (changed);
    return followSets;
}
```

构造 SLR(1) 分析表

采用教材上的算法 4.6 即可。

```
set<Item> Grammar::closure(const set<Item> &items) const {
    set<Item> newItems = items;
    bool changed = false;
    do {
        changed = false;
        for (const auto &item : newItems) {
            if (item.dotPos == (ssize_t)item.right.size()) {
                continue;
            }
            Symbol nextSymbol = item.right[item.dotPos];
            if (nextSymbol.isTerminal()) {
                continue;
            }
            for (const auto &right : rules_.at(nextSymbol)) {
                Item newItem{nextSymbol, right, 0, right.idx};
                if (find(newItems.begin(), newItems.end(), newItem) == newItems.
    end()) {
                    newItems.insert(newItem);
                    changed = true;
                }
            }
        }
    } while (changed);
    return newItems;
}

LR1Table Grammar::buildLR1Table() const {
    auto followSets = getAllFollowSets();

    vector<pair<Symbol, Phrase>> rules;
    for (const auto &[left, rightList] : rules_) {
        for (const auto &right : rightList) {
            rules.push_back({left, right});
```

```
33          }
34      }
35      LR1Table table(rules);
36      vector<ItemSet> itemSets;
37      map<set<Item>, int> items2Id;
38      queue<int> q;
39      auto startItem = Item{start_, rules_.at(start_)[0], 0, rules_.at(start_)[0].
        idx};
40      auto startItemSet = closure({startItem});
41      itemSets.push_back({startItemSet, {}});
42      items2Id[startItemSet] = 0;
43      q.push(0);
44      while (!q.empty()) {
45          int id = q.front();
46          q.pop();
47          map<Symbol, set<Item>> nextKernelItemSets;
48          for (const Item &item : itemSets[id].items) {
49              if (item.dotPos == (ssize_t)item.right.size()) {
50                  for (const auto &follow : followSets[item.left]) {
51                      if (item.left == start_) {
52                          if (!table.tryInsertAction({id, follow}, {Action::ACCEPT,
        0})) {
53                              printf("Not SLR(1) grammar!\n");
54                              exit(0);
55                          }
56                      } else {
57                          if (!table.tryInsertAction({id, follow}, {Action::REDUCE,
        item.id})) {
58                              printf("Not SLR(1) grammar!\n");
59                              exit(0);
60                          }
61                      }
62                  }
63              } else {
64                  Symbol nextSymbol = item.right[item.dotPos];
65                  Item newItem = {item.left, item.right, item.dotPos + 1, item.id};
66                  nextKernelItemSets[nextSymbol].insert(newItem);
67              }
68          }
69          for (const auto &[symbol, itemSet] : nextKernelItemSets) {
70              auto nextItemSet = closure(itemSet);
71              int nextId;
```

```
72              if (items2Id.find(nextItemSet) == items2Id.end()) {
73                  items2Id[nextItemSet] = itemSets.size();
74                  itemSets.push_back({nextItemSet, {}});
75                  q.push(itemSets.size() - 1);
76                  nextId = itemSets.size() - 1;
77              } else {
78                  nextId = items2Id[nextItemSet];
79              }
80              itemSets[id].to[symbol] = nextId;
81              if (symbol.isTerminal()) {
82                  if (!table.tryInsertAction({id, symbol}, {Action::SHIFT, nextId})
    ) {
83                      printf("Not SLR(1) grammar!\n");
84                      exit(0);
85                  }
86              } else {
87                  table.insertGoto({id, symbol}, nextId);
88              }
89          }
90      }
91      for (int i = 0; i < (ssize_t)itemSets.size(); i++) {
92          printf("I%d[label=\"I%d:\\n", i, i);
93          for (const auto &item : itemSets[i].items) {
94              std::cout << item << "\\n";
95          }
96          printf("\"]\n");
97          for (const auto &[symbol, dest] : itemSets[i].to) {
98              printf("I%d->I%d[label=\"%s\"]\n", i, dest, symbol.name().c_str());
99          }
100     }
101     return table;
102 }
```

LR 分析

采用教材上的算法 4.3 即可。

```
1  bool LR1Table::parse(std::function<Symbol()> getNext) const {
2      stack<int> st;
3      st.push(0);
4      int step = 0;
5      Symbol next = getNext();
6      while (!st.empty()) {
```

```
        auto top = st.top();
        auto it = action_.find({top, next});
        if (it == action_.end()) {
            return false;
        }
        auto action = it->second;
        if (action.type == Action::ACCEPT) {
            break;
        } else if (action.type == Action::SHIFT) {
            st.push(action.state);
            next = getNext();
        } else if (action.type == Action::REDUCE) {
            std::cout << "(" << ++step << ") ";
            auto rule = rules_[action.rule];
            std::cout << rule.first << " → " << rule.second << std::endl;
            for (int i = 0; i < (ssize_t)rule.second.size(); ++i) {
                st.pop();
            }
            st.push(goto_.at({st.top(), rule.first}));
        }
    }
    return next == Symbol::end;
}
```

3 测试

输入 1

```
(1+2)*(3+4)+5-(((6)/2))
```

输出 1

```
(1) F → num
(2) T → F
(3) E → T
(4) F → num
(5) T → F
(6) E → E+T
(7) F → (E)
(8) T → F
(9) F → num
(10) T → F
```

```
(11) E → T
(12) F → num
(13) T → F
(14) E → E+T
(15) F → (E)
(16) T → T*F
(17) E → T
(18) F → num
(19) T → F
(20) E → E+T
(21) F → num
(22) T → F
(23) E → T
(24) F → (E)
(25) T → F
(26) F → num
(27) T → T/F
(28) E → T
(29) F → (E)
(30) T → F
(31) E → T
(32) F → (E)
(33) T → F
(34) E → E-T
Accept!
```

程序给出了一个该输入的规范规约。

输入 2

```
1+2*(3)(4)
```

输出 2

```
(1) F → num
(2) T → F
(3) E → T
(4) F → num
(5) T → F
(6) F → num
(7) T → F
(8) E → T
Reject!
```

4 总结

这次实验让我熟悉了构造 LR 项目集规范族的方法和 LR 分析方法，也锻炼了编程能力和测试能力。