语法分析程序的设计与实现——LL 分析方法

任飞 2021210724

目录

1 实验题目

1.1 内容

编写 LL(1) 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. 编程实现算法 4.2，为给定文法自动构造预测分析表。
2. 编程实现算法 4.1，构造 LL(1) 预测分析程序。

2 程序设计说明

2.1 使用方法

使用 C++ 完整地实现了实验要求的功能。使用方法为：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消除左递归、提取左公因子后的产生式。如果文法不是 LL(1) 文法，程序会输出 `Not LL(1) grammar!` 并退出。如果文法是 LL(1) 文法，程序会输出构造的预测分析表。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 → TE'
E' → +TE'|-TE'|eps
F → (E)|num
T → (E)T'|numT'
T' → *FT'|/FT'|eps

     $          (          )          *          +          -          /          num
E               E->TE'                                                            E->TE'
E'   E'->eps                E'->eps               E'->+TE'   E'->-TE'
F               F->(E)                                                            F->num
T               T->(E)T'                                                          T->numT'
T'   T'->eps                T'->eps    T'->*FT'   T'->eps    T'->eps    T'->/FT'
```

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（即最左推导），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

程序由 `Grammar` 和 `LL1Table` 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` 类提供了消除左递归、消除左公因式、求 FIRST 集、求 FOLLOW 集、构造预测分析表的接口。

`map<pair<Symbol, Symbol>, Phrase>` 类型表示预测分析表，即 `LL1Table` 类。`LL1Table` 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

消除左递归

采用教材上的算法 2.1 消除左递归。

```
1  void Grammar::solveLeftRecursion() {
2      map<Symbol, vector<Phrase>> altRules;
3      for (auto &[left, rightList] : rules_) {
4          vector<Phrase> tempRightList;
5          // 代入前面已经消除左递归的产生式
6          for (const auto &right : rightList) {
7              if (right.empty()) {
8                  continue;
9              }
10             Symbol first = right[0];
11             if (!first.isTerminal() && first < left) {
12                 for (const auto &right2 : rules_[first]) {
13                     auto newRight = right2;
14                     newRight.insert(newRight.end(), right.begin() + 1, right.end
       ());
15                     tempRightList.push_back(newRight);
16                 }
17             } else {
18                 tempRightList.push_back(right);
19             }
20         }
21         // 消除左递归
22         Symbol altLeft = left.toAlternate();
23         vector<Phrase> newRightList, altRightList;
24         for (auto &right : tempRightList) {
25             if (right[0] == left) {
26                 right.erase(right.begin());
27                 right.push_back(altLeft);
28                 altRightList.push_back(std::move(right));
29             } else {
30                 right.push_back(altLeft);
```

```
                newRightList.push_back(std::move(right));
            }
        }
        if (!altRightList.empty()) {
            altRightList.push_back({});
            rightList = newRightList;
            altRules[altLeft] = altRightList;
        }
    }
    rules_.insert(altRules.begin(), altRules.end());
}
```

提取左公因子

维护一个队列，先将所有非终结符放入队列。对于每一个从队列中取出的非终结符，根据首字符对其右部进行分组，如果某一组中右部数量大于 1，则将其提取出来作为一个新的非终结符，将原来的产生式右部替换为新的非终结符，再将新的非终结符的产生式加入到文法中，并把新的非终结符加入到队列中。

```
void Grammar::solveLeftCommonFactor() {
    queue<Symbol> solveQueue;
    for (const auto &[left, rightList] : rules_) {
        solveQueue.push(left);
    }
    while (!solveQueue.empty()) {
        Symbol left = solveQueue.front();
        solveQueue.pop();
        vector<Phrase> &rightList = rules_[left];
        map<Symbol, vector<Phrase>> rightWithFirstMap;
        for (const auto &right : rightList) {
            if (!right.empty()) {
                rightWithFirstMap[right[0]].push_back(right);
            }
        }
        Symbol altLeft = left;
        for (auto &[first, rightWithFirstList] : rightWithFirstMap) {
            if (rightWithFirstList.size() > 1) {
                altLeft = altLeft.toAlternate();
                auto &altLeftRightList = rules_[altLeft];
                for (auto &right : rightWithFirstList) {
                    altLeftRightList.emplace_back(right.begin() + 1, right.end())
    ;
                    rightList.erase(std::find(rightList.begin(), rightList.end(),
```

```
    right));
                }
                rightList.push_back({first, altLeft});
                solveQueue.push(altLeft);
            }
        }
    }
}
```

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。

```
set<Symbol> Grammar::getFirstSet(const Phrase &phrase) const {
    auto iter = firstSetCache_.find(phrase);
    if (iter != firstSetCache_.end()) {
        return iter->second;
    }
    firstSetCache_[phrase] = {}; // 避免无穷递归
    set<Symbol> firstSet;
    bool allHaveEpsilon = true;
    for (const auto &symbol : phrase) {
        if (symbol.isTerminal()) {
            firstSet.insert(symbol);
            allHaveEpsilon = false;
            break;
        } else {
            bool haveEpsilon = false;
            for (const auto &right : rules_.at(symbol)) {
                auto s = getFirstSet(right);
                if (s.find(Symbol::epsilon) != s.end()) {
                    s.erase(Symbol::epsilon);
                    haveEpsilon = true;
                }
                firstSet.merge(s);
            }
            if (!haveEpsilon) {
                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;
}
```

构造预测分析表

采用教材上的算法 4.2 即可。

```
LL1Table Grammar::buildLL1Table() const {
    LL1Table table{start_};
    auto followSets = getAllFollowSets();
    for (const auto &[left, rightList] : rules_) {
        for (const auto &right : rightList) {
            auto firstSet = getFirstSet(right);
            for (const auto &symbol : firstSet) {
                if (symbol == Symbol::epsilon) {
                    for (const auto &symbol2 : followSets.at(left)) {
                        if (!table.tryInsert({left, symbol2}, right)) {
                            std::cout << "Not LL(1) grammar!" << std::endl;
                            exit(0);
                        }
                    }
                } else {
                    if (!table.tryInsert({left, symbol}, right)) {
                        std::cout << "Not LL(1) grammar!" << std::endl;
                        exit(0);
                    }
                }
            }
        }
    }
    return table;
}
```

2.4 预测分析

采用教材上的算法 4.1 即可。

```
bool LL1Table::parse(std::function<Symbol()> getNext) const {
    stack<Symbol> st;
    int step = 0;
    st.push(start_);
    Symbol next = getNext();
    while (!st.empty()) {
```

```
        auto top = st.top();
        st.pop();
        if (top.isTerminal()) {
            if (top == next) {
                next = getNext();
                continue;
            } else {
                return false;
            }
        } else {
            if (table_.find({top, next}) == table_.end()) {
                return false;
            }
            auto right = table_.at({top, next});
            std::cout << "(" << ++step << ") " << top << " → " << right << std::
    endl;
            for (auto it = right.rbegin(); it != right.rend(); ++it) {
                st.push(*it);
            }
        }
    }
    return next == Symbol::end;
}
```

3 测试

3.1 算术表达式的语法分析

首先测试实验要求的对算术表达式文法的语法分析。

输入 1

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

输出 1

```
(1) E → TE'
(2) T → (E)T'
(3) E → TE'
(4) T → numT'
(5) T' → eps
(6) E' → +TE'
```

```
(7) T → numT'
(8) T' → eps
(9) E' → eps
(10) T' → *FT'
(11) F → (E)
(12) E → TE'
(13) T → numT'
(14) T' → eps
(15) E' → +TE'
(16) T → numT'
(17) T' → eps
(18) E' → eps
(19) T' → eps
(20) E' → +TE'
(21) T → numT'
(22) T' → eps
(23) E' → -TE'
(24) T → (E)T'
(25) E → TE'
(26) T → (E)T'
(27) E → TE'
(28) T → (E)T'
(29) E → TE'
(30) T → numT'
(31) T' → eps
(32) E' → eps
(33) T' → /FT'
(34) F → num
(35) T' → eps
(36) E' → eps
(37) T' → eps
(38) E' → eps
(39) T' → eps
(40) E' → eps
Accept!
```

程序给出了一个该输入的最左推导。

输入 2

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

输出 2

```
(1) E → TE'
(2) T → numT'
(3) T' → eps
(4) E' → +TE'
(5) T → numT'
(6) T' → *FT'
(7) F → (E)
(8) E → TE'
(9) T → numT'
(10) T' → eps
(11) E' → eps
Reject!
```

3.2 文法变换测试

为了验证文法变换具有通用性，而不是仅仅适用于题目给定的文法，给出如下具有间接左递归和左公因式的文法：

$$A \to Ba|a|b|bc$$
$$B \to BA|Bc|c|Ad$$

使用代码表示为

```
grammar.addRule("A", {"B", "a"});
grammar.addRule("A", {"a"});
grammar.addRule("A", {"b"});
grammar.addRule("A", {"b", "c"});
grammar.addRule("B", {"B", "A"});
grammar.addRule("B", {"B", "c"});
grammar.addRule("B", {"c"});
grammar.addRule("B", {"A", "d"});
grammar.setStart("A");
```

输出为

```
A → Ba|a|b|bc
B → BA|Bc|c|Ad

A → Ba|a|bA'
A' → eps|c
B → cB'|adB'|bB'
B' → AB'|adB'|eps|dB'|cB''
B'' → B'|dB'

```

```
Not LL(1) grammar!
```

程序正确地完成了消除左递归和提取左公因式的工作，并且检测到了该文法不是 LL(1) 文法。

4 总结

这次实验让我熟悉了各种文法变换算法，深入理解了 LL1 预测分析方法，也锻炼了编程能力和测试能力。