一、实验目的

1.掌握左递归消去的方法

2.掌握提取左公共因子的方法

二、 实验内容与实验要求

二、实验内容

1.输入任意的上下文无关文法，输出一个提取了左公因子且消除了左递归且的等价文法；

2 编制测试程序；

3.调试程序

三、实验要求

1.采用任意语言，实现该算法；

2.上交源代码；

3.撰写实验报告，在实验报告里要写清楚测试结果，并以 word 形式上交实验报告；

三、 设计方案与算法描述

消除左递归采用书本的方法

提取左公因子采用构建字典树的方法

消除左递归

消除直接左递归的方法:

1、把所有产生式写成候选式形式。如 A→Aa1｜Aa2……｜Aan｜b1｜b2……｜bm。其中每个 a 都不等于ε，而第个 b 都不以 A 开头。

2、变换候选式成如下形式：

A→b1A’｜b2A’……｜bmA’

A’ →a1A’｜a2A’……｜anA’｜ε

对于间接左递归先改写产生式子再去消除直接左递归

```
for i in 0..n {
    for j in 0..i {
      把所有以 Aj 开头的 Ai 的产生式替换为 Aj 的产生式子
    }
    消除 Ai 的直接左递归
}
```

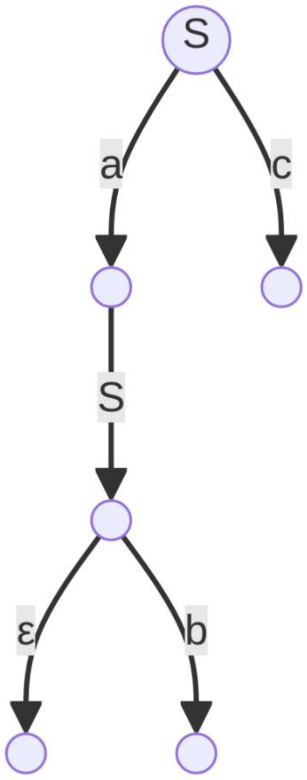
提取左公因子

如：以下的文法可以表示成一颗字典树

S->aSb

S->aS

S->c



如图这棵树的边对应产生式中的一个符号

叶子对应产生式的结尾

有一个以上的分叉的结点暗示有对应的公共左因子

提取左公因子时自底向上地进行

如果该节点是叶子节点那么对应一个产生式的结尾所以返回一个空字符串

如果该节点有且仅有一个孩子说明根到该节点对应的左因子不是公共左因子因此返回到孩子边上的符号加上孩子返回的结果

如果该节点有两个或以上的孩子。 那么分配一个新非终结符。 新非终结符的产生式就是孩子们返回的结果。 然后返回分配到的非终结符

四、 测试结果

测试例子

start: S

nonterminals: S

S->aSb

S->aS

S->c

start: E

nonterminals: ET

E->E+T|T

start: S

nonterminals: SAB

S->Ac|c

A->Bb|b

B->Sa|a

start: E

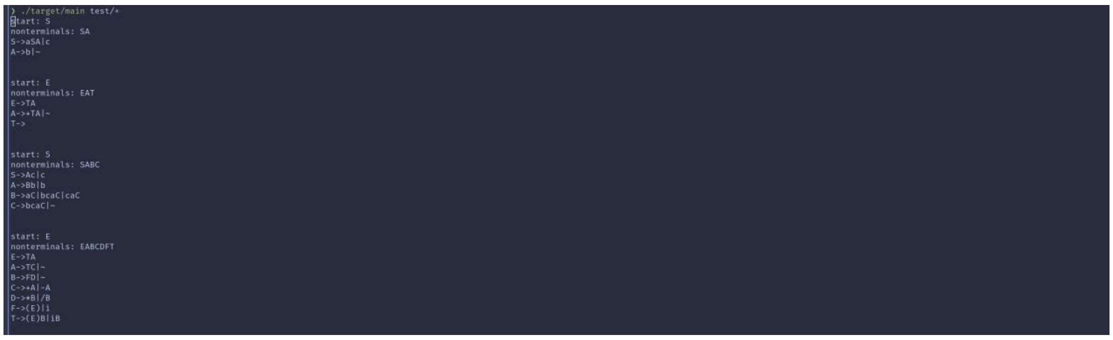
nonterminals: ETF

E->ET+|ET-|T

T->TF*|TF/|F

F->(E)|i

结果



```
❯ ./target/main test/*
start: S
nonterminals: SA
S->aSA|c
A->b|~


start: E
nonterminals: EAT
E->TA
A->+TA|~
T->


start: S
nonterminals: SABC
S->Ac|c
A->Bb|b
B->aC|bcaC|caC
C->bcaC|~


start: E
nonterminals: EABCDFT
E->TA
A->TC|~
B->FD|~
C->+A|-A
D->*B|/B
F->(E)|i
T->(E)B|iB
```

五、　源代码

```
#ifndef CFG_HPP

#define CFG_HPP



#include <bitset>

#include <cassert>

#include <map>

#include <optional>

#include <string>

#include <unordered_map>

#include <vector>



using std::bitset;

using std::map;
using std::optional;

using std::string;

using std::string_view;

using std::unordered_map;

using std::vector;


// 解析不允许有任何空格符

// 1. 产生式的左部和右部用 "->" 分隔

// 2. 右部的多个候选项用 "|" 分隔

// 3. 终结符用单个小写字母表示

// 4. 非终结符用单个大写字母表示

// 5. ~ 表示空产生式

struct ContextFreeGrammar {

  static constexpr int MAX_TERMINAL_NUM = 26;

  static constexpr int MAX_NONTERMINAL_NUM = 26;

  using Symbol = char;

  using ProductionRight = string;

  using ProductionRights = vector<string>;

  using Productions = unordered_map<Symbol, ProductionRights>;

  static constexpr Symbol EPSILON = '~';

  static constexpr char DIVISION = '|';

  static constexpr char END = '$';
  static constexpr string_view ARROW = "->";
  static constexpr string_view START_PREFIX = "start: ";
  static constexpr string_view NONTERMINAL_SET_PREFIX = "nonterminals: ";
  static constexpr string_view TERMINAL_SET_PREFIX = "terminals: ";

  static bool is_terminal(Symbol symbol) {
      return !is_nonterminal(symbol) && !is_epsilon(symbol);
  };
  static bool is_nonterminal(Symbol symbol) {
      return symbol >= 'A' && symbol <= 'Z';
  }
  static bool is_epsilon(Symbol symbol) { return symbol == EPSILON; }

  struct NonterminalSet : bitset<MAX_NONTERMINAL_NUM> {
      NonterminalSet() {}
      NonterminalSet(string_view nonterminals) {
          bitset<MAX_NONTERMINAL_NUM>::reset();
          for (auto symbol : nonterminals) {
              assert(is_nonterminal(symbol));
              this->set(symbol);
          }
      }
      bool get(Symbol symbol) { return (*this)[symbol - 'A']; }
      void set(Symbol symbol) { bitset<MAX_NONTERMINAL_NUM>::set(symbol -
  'A'); }
      void reset(Symbol symbol) {
        bitset<MAX_NONTERMINAL_NUM>::reset(symbol - 'A');
      }
      string to_string(Symbol start) {
        auto ret = string{start};
        for (Symbol s = 'A'; s <= 'Z'; s++) {
          if (this->get(s) && s != start) {
            ret += s;
          }
        }
        return ret;
      }

      void alloc(Symbol symbol) {
        assert(is_nonterminal(symbol));
        assert(!this->get(symbol));
        this->set(symbol);
      }
      Symbol alloc() {
          for (auto i = 'A'; i <= 'Z'; i++) {
              if (!this->get(i)) {
                  this->set(i);
                  return i;
              }
          }
          assert(false);
      }
  };


  ContextFreeGrammar(Symbol start, string_view nonterminals)
          : _nonterminals(nonterminals) {
      this->_start = start;
      for (auto non_term : nonterminals) {
          this->_productions.insert({non_term, {}});
      }
  }
  ContextFreeGrammar(Symbol start, NonterminalSet nonterminals,
                                      Productions productions)
          : _start(start), _nonterminals(nonterminals), _productions(productions) {}
```

```
ProductionRights &produce(Symbol nonterminal) {
    assert(is_nonterminal(nonterminal) && "expect nonterminal symbol");
    return _productions.at(nonterminal);
}

// 第一项一定是起始符号
string nonterminals() { return this->_nonterminals.to_string(this->_start); }

Symbol start() { return this->_start; }

string to_string() {
    auto ret = string(START_PREFIX) + this->_start + '\n';
    auto nonterminals = this->nonterminals();
    ret += string(NONTERMINAL_SET_PREFIX) + nonterminals + '\n';
    for (auto left : nonterminals) {
        ret += left;
        ret += ARROW;
        auto &rights = this->produce(left);
        for (int i = 0; i < rights.size(); i++) {
            if (i != 0) {
                ret += DIVISION;
            }
            ret += rights.at(i);
        }
        ret += '\n';
    }
    return ret;
}

Symbol alloc_nonterminal() {
    auto new_non = this->_nonterminals.alloc();
    this->_productions.insert({new_non, {}});
    return new_non;
}

void alloc_nonterminal(Symbol nonterm) {
    this->_nonterminals.alloc(nonterm);
    this->_productions.insert({nonterm, {}});
}

ContextFreeGrammar clone() {
    auto productions = this->_productions;
    auto start = this->_start;
    auto non = this->_nonterminals;
    return ContextFreeGrammar(start, non, productions);
  }

private:
  NonterminalSet _nonterminals;
  Symbol _start;
  Productions _productions;
};

#endif
#include "../../common/CFG.hpp"
#include <iostream>
#include <optional>

void rewrite(ContextFreeGrammar &cfg, ContextFreeGrammar::Symbol a_i,
             ContextFreeGrammar::Symbol a_j);

void handle_direct(ContextFreeGrammar &cfg,
                   ContextFreeGrammar::Symbol to_handle);

void handle_epsilon(ContextFreeGrammar &cfg) {
  auto nonterminals = cfg.nonterminals();
```

```
for (auto noterm : nonterminals) {
    auto &rights = cfg.produce(noterm);
    for (int i = 0; i < rights.size(); i++) {
        auto right = rights.at(i);
        if (right.size() == 1 && right.front() == ContextFreeGrammar::EPSILON) {
            rights.erase(rights.begin() + i);
            for (auto to_handle : nonterminals) {
                if (to_handle != noterm) {
                    auto &rights = cfg.produce(to_handle);
                    auto size = rights.size();
                    for (int i = 0; i < size; i++) {
                        if (rights.at(i).find(noterm) != string::npos) {
                            auto right = rights.at(i);
                            auto pos = right.find(noterm);
                            while (pos != string::npos) {
                                right.erase(pos, 1);
                                pos = right.find(noterm);
                            }
                            if (!right.empty()) {
                                rights.push_back(right);
                            }
                        }
                    }
                }
            }
            break;
        }
    }
}
}

std::optional<ContextFreeGrammar::Symbol>
find_epsilon(ContextFreeGrammar &cfg,
                                                        string_view nons) {
    for (auto non : nons) {
        auto &rights = cfg.produce(non);
        for (int i = 0; i < rights.size(); i++) {
            auto right = rights.at(i);
            if (right.size() == 1 && right.front() == ContextFreeGrammar::EPSILON) {
                rights.erase(rights.begin() + i);
                return {non};
            }
        }
    }
```

```
  return {};
}

void remove_(ContextFreeGrammar &cfg, string_view nons,
             ContextFreeGrammar::Symbol to_remove) {
  for (auto to_handle : nons) {
  }
}

// 不能推导出环
void left_recursion_kill(ContextFreeGrammar &cfg) {
  auto nonterminals = cfg.nonterminals();
  auto size = nonterminals.size();
  for (int i = 0; i < size; i++) {
    auto a_i = nonterminals.at(i);
    for (int j = 0; j < i; j++) {
      auto a_j = nonterminals.at(j);
      rewrite(cfg, a_i, a_j);
    }
    handle_direct(cfg, a_i);
  }
}
```

```
// 消除直接左递归
// 消除后可能会增加一个新的非终结符
void handle_direct(ContextFreeGrammar &cfg,
                   ContextFreeGrammar::Symbol to_handle) {
  auto left_recursion = ContextFreeGrammar::ProductionRights();
  auto no_left_recursion = ContextFreeGrammar::ProductionRights();
  for (auto right : cfg.produce(to_handle)) {
    if (right.front() == to_handle) {
      left_recursion.push_back(right);
    } else {
      no_left_recursion.push_back(right);
    }
  }
  if (!left_recursion.empty()) {
    auto new_nonterm = cfg.alloc_nonterminal();
    for (auto &entry : no_left_recursion) {
      entry.push_back(new_nonterm);
    }
    for (auto &entry : left_recursion) {
      entry.erase(entry.begin());
      entry.push_back(new_nonterm);
    }
```

```
        left_recursion.push_back({ContextFreeGrammar::EPSILON});
        cfg.produce(to_handle) = no_left_recursion;
        cfg.produce(new_nonterm) = left_recursion;
    }
}

// 处理 Ai -> Ajγ
void rewrite(ContextFreeGrammar &cfg, ContextFreeGrammar::Symbol a_i,
             ContextFreeGrammar::Symbol a_j) {
    auto new_rights = ContextFreeGrammar::ProductionRights();
    for (auto right : cfg.produce(a_i)) {
        if (right.front() == a_j) {
            right.erase(right.begin());
            for (auto prefix : cfg.produce(a_j)) {
                prefix.insert(prefix.end(), right.begin(), right.end());
                new_rights.push_back(prefix);
            }
        } else {
            new_rights.push_back(right);
        }
    }
    cfg.produce(a_i) = new_rights;
```

```
}
#include "../../common/CFG.hpp"

using std::map;

struct TrieNode : map<ContextFreeGrammar::Symbol, TrieNode *> {
  TrieNode(ContextFreeGrammar &cfg, ContextFreeGrammar::Symbol symbol)
      : map<ContextFreeGrammar::Symbol, TrieNode *>() {
    for (auto &right : cfg.produce(symbol)) {
      auto cur = this;
      auto it = right.begin();
      for (;;) {
        // 迭代直到遍历完右部或者在树上找不到对应的节点
        if (it == right.end()) {
          cur->insert({ContextFreeGrammar::EPSILON, new TrieNode});
          break;
        } else if (auto entry = cur->find(*it); entry == cur->end()) {
          if (cur->size() == 0 && cur != this) {
            cur->insert({ContextFreeGrammar::EPSILON, new TrieNode});
          }
          break;
        } else {
                cur = entry->second;
                it++;
            }
        }
        while (it != right.end()) {
            auto new_child = new TrieNode;
            cur->insert({*it, new_child});
            cur = new_child;
            it++;
        }
    }
}

TrieNode() : map<ContextFreeGrammar::Symbol, TrieNode *>() {}
ContextFreeGrammar::ProductionRight walk(ContextFreeGrammar &cfg) {
    if (this->size() == 0) {
        return "";
    } else if (this->size() == 1) {
        auto [symbol, child] = *this->cbegin();
        return symbol + child->walk(cfg);
    } else {
        auto new_nonterm = cfg.alloc_nonterminal();
```

```
            auto &rights = cfg.produce(new_nonterm);
            for (auto [symbol, child] : *this) {
                rights.push_back(symbol + child->walk(cfg));
            }
            return ContextFreeGrammar::ProductionRight{new_nonterm};
        }
    }
};

struct TrieTree {
    ContextFreeGrammar &cfg;
    ContextFreeGrammar::Symbol symbol;
    TrieNode root;
    TrieTree(ContextFreeGrammar &cfg, ContextFreeGrammar::Symbol symbol)
            : cfg(cfg), symbol(symbol), root(cfg, symbol){};

    void extract_left_factor() {
        auto &rights = cfg.produce(symbol);
        rights.clear();
        for (auto [symbol, child] : this->root) {
            rights.push_back(symbol + child->walk(cfg));
        }
    }
};


void extract_left_factor(ContextFreeGrammar &cfg) {
    auto non_terms = cfg.nonterminals();
    for (auto non_term : non_terms) {
        TrieTree(cfg, non_term).extract_left_factor();
    }
}
#include "../common/CFG.hpp"
#include "../common/CfgParser.hpp"
#include "lf/lf.h"
#include "lrk/lrk.h"
#include <cassert>
#include <cstdio>
#include <iostream>


int main(int argc, char *argv[]) {
    assert(argc > 1);
    for (int i = 1; i < argc; i++) {
        auto file = argv[i];
        auto cfg = CfgParser(Util::read_file_to_string(file)).parse();
        left_recursion_kill(cfg);

        extract_left_factor(cfg);

        std::cout << cfg.to_string() << std::endl;

        getchar();

    }

    return 0;

}
```