一、实验目的

1.掌握 First 集的求解方法

2.掌握 Follow 集的求解方法

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，输出各非终结符的 First 集、Follow 集；

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

First 集求法

```
初始化 firsts = { x => {} for x in 非终结符集合 }
{
    for x in 非终结符集合 {
        let first = firsts.entry(x)
        for right in x 的推导式 {
            for symbol in right {
                let set = match symbol {
                    epsilon => {ε},
                    非终结符  non => firsts.get(non),
                    终结符  term => {term}
                }
                if set.contain_and_remove_epsilon() {
                    first += set
                } else {
                    first += set
                    break
                }
            }
            if for 循环没有被 break {
                first += {ε}
            }
        }
    }
} until firsts  不再变化
```

follow 集合求解

```
初始化  follows := { x => {} for x in  非终结符集合  且  x 不为开始符号, Start => {$} }
{
    for (left, right) in  所有产生式  {
            for i in right {
                for j in 从 i 到产生式结尾的子串（不包括 i） {
                    match j {
                        ε => conutine,
                        终结符 term => follows.at(term).add(term),
                        非终结符 non => {
                            let first = firsts.get(non)
                            if first.contains_and_remove_epsilon() {
                                follows.at(term).add(first)
                            } else {
                                follows.at(term).add(first)
                                break
                            }
                        }
                    }
                    if 上面的 for 循环没有被 break {
                        follows.at(i).add(follows.at(left))
                    }
                }
            }
        }
} until follows 不再变化
```

四、 测试结果

测试例子

start: S

nonterminals: SAB

S->A|a

A->Bc|~

B->d|AS

start: S

nonterminals: SHKLM

S->MH|a

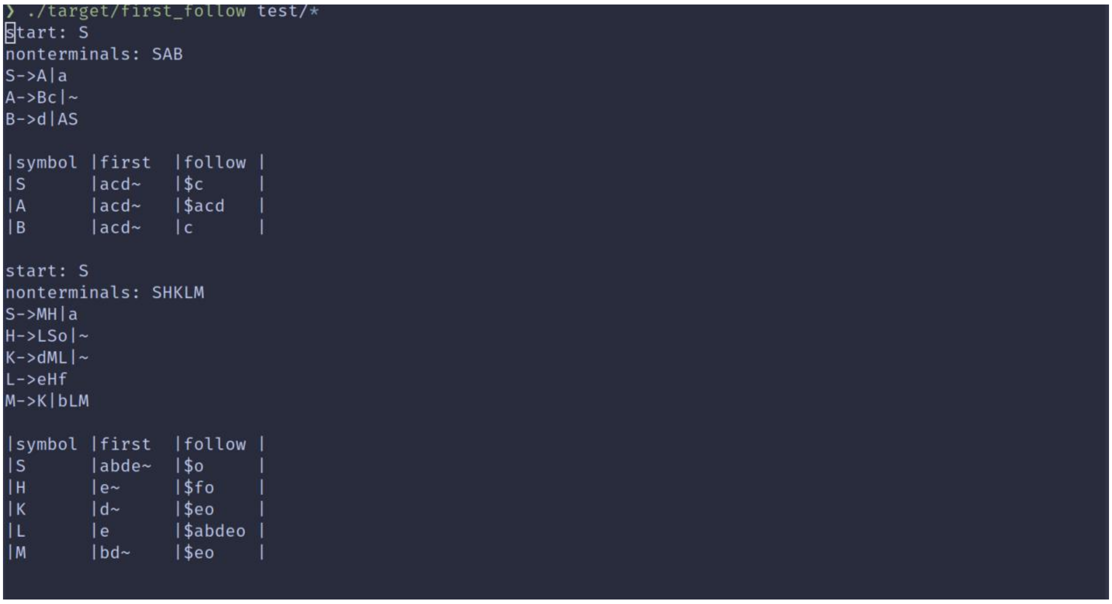
H->LSo|~

K->dML|~

L->eHf

M->K|bLM

结果



```
❯ ./target/first_follow test/*
start: S
nonterminals: SAB
S->A|a
A->Bc|~
B->d|AS

|symbol |first  |follow |
|S      |acd~   |$c     |
|A      |acd~   |$acd   |
|B      |acd~   |c      |

start: S
nonterminals: SHKLM
S->MH|a
H->LSo|~
K->dML|~
L->eHf
M->K|bLM

|symbol |first  |follow |
|S      |abde~  |$o     |
|H      |e~     |$fo    |
|K      |d~     |$eo    |
|L      |e      |$abdeo |
|M      |bd~    |$eo    |
```

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 stri    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 "./first_follow.h"
#include "../../common/CFG.hpp"
#include <algorithm>
#include <iostream>
#include <unordered_map>
#include <unordered_set>

SymbolSet first(ContextFreeGrammar &cfg, Map &firsts,
                const ContextFreeGrammar::ProductionRight &right);
bool first(ContextFreeGrammar &cfg, Map &firsts,
           ContextFreeGrammar::Symbol symbol);
```

```
}

SymbolSet first(ContextFreeGrammar &cfg, Map &firsts,
                const ContextFreeGrammar::ProductionRight &right) {
  auto ret = SymbolSet();
  auto i = 0;
  for (; i < right.size(); i++) {
    auto symbol = right.at(i);
    auto set = SymbolSet();
    if (ContextFreeGrammar::is_terminal(symbol)) {
      set.set(symbol);
    } else if (ContextFreeGrammar::is_nonterminal(symbol)) {
      set = firsts.at(symbol);
    } else if (ContextFreeGrammar::is_epsilon(symbol)) {
      set.add_epsilon();
    }
    if (set.contains_and_remove_epsilon()) {
      ret |= set;
    } else {
      ret |= set;
      break;
    }
  }
```

```
    if (i == right.size()) {
        ret.add_epsilon();
    }
    return ret;
}

bool first(ContextFreeGrammar &cfg, Map &firsts,
                      ContextFreeGrammar::Symbol symbol) {
    auto &old = firsts.at(symbol);
    auto new_ = SymbolSet();
    for (const auto &right : cfg.produce(symbol)) {
        new_ |= first(cfg, firsts, right);
    }
    auto ret = new_ != old;
    old = new_;
    return ret;
}

Map solve_firsts(ContextFreeGrammar &cfg) {
    auto ret = Map();
    auto nonterminals = cfg.nonterminals();
    for (auto symbol : nonterminals) {
        ret.insert({symbol, {}});
    }
    for (;;) {
        auto flag = false;
        for (auto symbol : nonterminals) {
            flag |= first(cfg, ret, symbol);
        }
        if (!flag) {
            break;
        }
    }
    return ret;
}


Map solve_follows(ContextFreeGrammar &cfg, Map &firsts) {
    auto nonterminals = cfg.nonterminals();
    auto start = cfg.start();
    auto follows = Map{{start, {ContextFreeGrammar::END}}};
    for (auto nonterm : nonterminals) {
        follows.insert({nonterm, {}});
    }
```

```
for (;;) {
    auto flag = false;
    for (auto nonterm : nonterminals) {
        auto &follow = follows.at(nonterm);
        for (auto &right : cfg.produce(nonterm)) {
            for (auto i = 0; i < right.size(); i++) {
                if (auto cur = right.at(i); ContextFreeGrammar::is_nonterminal(cur)) {
                    auto &follow_i = follows.at(cur);
                    auto j = i + 1;
                    for (; j < right.size(); j++) {
                        auto symbol = right.at(j);
                        if (ContextFreeGrammar::is_nonterminal(symbol)) {
                            auto first_j = firsts.at(symbol);
                            if (first_j.contains_and_remove_epsilon()) {
                                auto new_follow = follow_i | first_j;
                                flag |= new_follow != follow_i;
                                follow_i = new_follow;
                            } else {
                                auto new_follow = follow_i | first_j;
                                flag |= new_follow != follow_i;
                                follow_i = new_follow;
                                break;
                            }
                        } else if (ContextFreeGrammar::is_epsilon(symbol)) {
                            continue;
                        } else {
                            auto new_follow = follow_i | SymbolSet{symbol};
                            flag |= new_follow != follow_i;
                            follow_i = new_follow;
                            break;
                        }
                    }
                    if (j == right.size()) {
                        auto new_follow = follow_i | follow;
                        flag |= new_follow != follow_i;
                        follow_i = new_follow;
                    }
                }
            }
        }
    }
    if (!flag) {
        break;
    }
```

```
    }

    return follows;
}
#include "../../03-cfg-trans/lrk/lrk.h"
#include "../../common/CfgParser.hpp"
#include "../../common/comm.hpp"
#include "./first_follow.h"
#include <algorithm>
#include <cassert>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <string>
#include <string_view>

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();
        auto nonterminals = cfg.nonterminals();
        std::cout << cfg.to_string() << std::endl;
        std::cout << "|symbol\t|first\t|follow\t|" << std::endl;
        auto firsts = solve_firsts(cfg);
        auto follows = solve_follows(cfg, firsts);
        for (auto symbol : nonterminals) {
            std::cout << "|" << symbol << "\t";
            std::cout << "|" << firsts.at(symbol).to_string() << "\t";
            std::cout << "|" << follows.at(symbol).to_string() << "\t|" << std::endl;
        }
        getchar();
    }
    return 0;
}
```