# 算符优先语法分析设计

## 程序功能描述

输入算符优先文法，根据算符优先文法的特点构造firstvt集、lastvt集和算符优先矩阵。给定词法分析输出的中间文件，根据算符优先分析法判断是否符合语法规则。

## 数据结构描述

本实验的数据结构与专题3的数据结构十分接近，如下所示。

本实验包含如下数据结构：firstvt集、lastvt集、文法集合、算符优先矩阵、非终结符号集和终结符号集。

首先说明firstvt集与lastvt集的类型是Set。在OpFirst.h中定义此类型如下，

typedef map<char, set<char>> Set;

这里的map是C++的映射类型，采用key-value结构存储数据，这里的含义是以非终结符号作为key去寻找此非终结符号的firstvt集或lastvt集，而集合用set类型描述。

文法集合grammar的类型是NODES，定义如下，

typedef unordered\_map<char, set<string>> NODES;

这里使用了C++的无序映射类型，通过非终结符号找到其所有产生式右部，这些产生式用set<string>来存储。set<string>是一个集合类型，里面每一个元素类型均为string，即产生式的右部。

算符优先矩阵matrix的类型为OpFirstMatrix，定义如下，

typedef map<Query, int, compareQuery> LL1\_table;

这里的含义是通过Query类型的变量去查询两个终结符号的优先关系。Query类型定义如下，

typedef struct query {

char end1;//终结符号1

char end2;//终结符号2

}Query;

含义是通过终结符号1与终结符号2连接的查询二者之间的优先关系。若查询的结果为1，说明end1 > end2; 若结果为0，则end1 = end2; 若结果为-1，则end1 < end2。compareQuery是一个仿函数，用于map类型排序的方式，属于C++的语言规范，这里不展开讨论。

分析栈opFirst\_stack类型是vector<char>，采用C++的容器类型，每一个元素类型为char。

非终结符号集nonTerminate与终结符号集Terminate的类型是set<char>，分别用于存储符号。

此外还有一个类型为NODE，定义如下，

//产生式

typedef struct node {

char left;//产生式左部

string right;//产生式右部

}NODE;

这里数据结构的定义充分体现了在利用算符优先矩阵进行归约的特点。

## 程序结构描述

## 3.1设计方法

程序遵循算符优先分析方法设计，分为三步走。1. 根据文法构造firstvt、lastvt集；2.根据上述两个集合构造算符优先矩阵；3. 利用算符优先矩阵和分析栈进行算符优先分析。文法如下所示，

G[E]:

E->E+T

E->E-T

E->T

T->T\*F

T->T/F

T->F

F->(E)

F->i

## 3.2程序模块

程序包含OpFirst.cpp、OpFirst.h、main.cpp、utility.cpp、lex.h。其中OpFirst.cpp负责算符优先分析，OpFirst.h定义了相关数据结构，main.cpp负责调用OpFirst.cpp的相关函数以及实施OpFirst分析。utility.cpp包含辅助函数。lex.h中定义了各种符号的宏。

## 3.3函数说明

OpFirst.cpp中主要包含getCurrent、splitGeneration、setGrammar、genFirstvt、genLastvt、genOpFirstMatrix、showFirstvt、showLastvt、showGrammar、showOpFirstMatrix、OpFirst这些函数。下面一一介绍。

getCurrent函数在第二次实验中用到，作用是从词法分析输出的中间文件提取出当前的单词符号，这里不再赘述。

splitGeneration函数的作用是分解产生式，传入参数是string类型的产生式generation，格式为E->AX，函数的返回值是NODE类型，将E->AX结构分解为E和AX，装入一个NODE变量返回。分解的方法是取generation的首字符作为左部，’->’之后的以首个非空字符作为首字符的子串为右部。

setGrammar函数的作用是设置文法NODES，同时收集非终结符号与终结符号。当用户在终端上输入一个’E->AX’结构后，setGrammar函数会先调用splitGeneration函数分解产生式(注：这里为了简单起见，输入产生式之间不允许有空格)。接下来，以产生式左部作为key，将相应的右部插入grammar中，写法如下，

grammar[gram.left].insert(gram.right);

然后将左部放入非终极符号集nonTerminate中，右部中的终结符号放入终结符号集Terminate中，写法如下，

nonTerminate.insert(gram.left);

genFirstvt函数负责生成非终结符号的firstvt集。分为两种情况：1). U -> b... 或 U -> Vb..., 则b 属于 firstvt(U)；2). U -> V..., 则firstvt(V) 在 firstvt(U)中。首先设立一个标志位change，作为firstvt集是否还增大的标志，若change为true则循环进行求解，否则退出。在循环里遍历文法，找出非终结符号的所有右部，针对某一个右部，判断首字符是否为终结符号，若是则满足第一种情况，将其加入当前非终结符号的firstvt集中。否则判断第二个符号是否为终结符号(如果有的话)，若是则同样满足第一种情况，将其加入firstvt集，否则将首字符的firstvt集加入到firstvt集中。

genLastvt函数负责生成非终结符号的lastvt集。分为两种情况：1). U -> ...a or U -> ...aQ；2). U -> ...Q, lastvt(Q) -> lastvt(U)。首先设立一个标志位change，作为lastvt集是否还增大的标志，若change为true则循环进行求解，否则退出。在循环里遍历文法，找出非终结符号的所有右部，针对某一个右部，首先判断末尾字符是否为终结符号，若是则符合第一种情况，将其加入lastvt集中；否则判断倒数第二个字符是否为终结符号，若是则同样符合第一种情况，将其加入lastvt集中。否则，将末尾非终结符号的lastvt集加入lastvt集中。

genOpFirstMatrix函数负责生成算符优先矩阵。首先处理#，# < firstvt(E), lastvt(E) > #，处理的方式很简单，用#和集合中的元素组合成一个Query变量q，将matrix[q]设立为-1、0、1三者之一。分为U -> ...ab...、U -> ...aA..、U -> ...aAb...、U -> ...Ab...这四种情况讨论。第一种情况，a与b的优先级相同；第二种情况，a小于firstvt(A)；第三种情况，a与b的优先级相同；第四种情况，lastvt(A)大于a。另外，#与#的优先级相同。

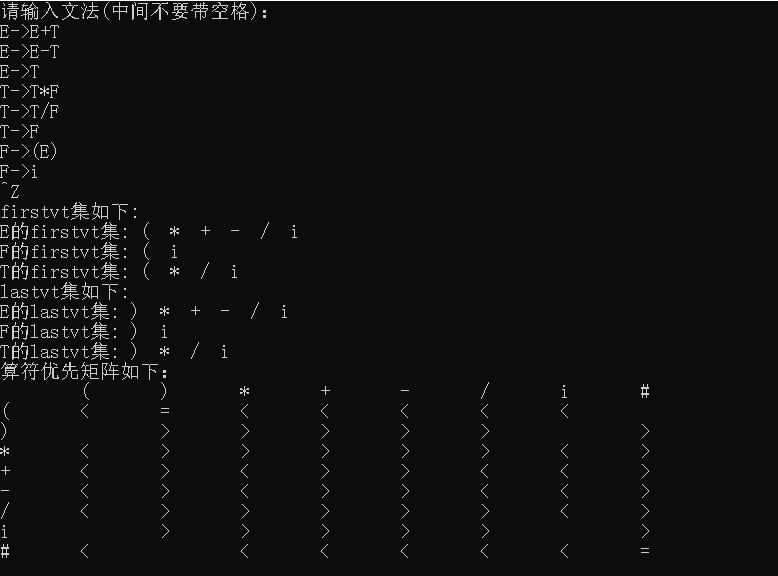
OpFirst函数负责算符优先分析。在函数开始定义了四个重要的变量：topEnd、topEndPos、inputCh和reduceCh。topEnd的含义是当前栈中最靠前的终结符号(归约串的最右边的终结符号)，topEndPos记录topEnd在栈中的位置，inputCh为当前输入符号，redeuceCh是归约符号。接下来进入循环，首先判断当前输入符号与topEnd的优先级，如果topEnd的优先级低，则将输入符号入栈，同时更新topEnd及topEndPos，输入串后移一位；否则，说明句柄已经在栈顶形成，首先判断栈顶符号是否为topEnd，若是则直接将其传入reduce函数获取归约符号，然后删除栈顶符号同时将新符号入栈，输入串后移一位；若栈顶符号不是topEnd，意味着句柄的左边界仍需要去找。方法是从topEnd的左边位置向前搜索第一个且满足优先级小于其后一个终结符号的终结符号，将找到的终结符号的右边作为句柄传入reduce函数，剩下的过程同上面一样。

在算符优先分析中用到了reduce函数，这个函数的作用是给定字符串，遍历文法找出符合条件的产生式左部。这里，符合条件指的是相同位置上的字符要么是相同的终结符号，要么均为非终结符号，同时长度一样，不能有其余情况。实现匹配的函数为match。

关于显示firstvt、lastvt、算符优先矩阵的函数与上一个实验类似，这里不再赘述。

## 程序测试

首先输入文法，获取firstvt、lastvt集和算符优先矩阵，如下所示，



然后根据中间文件(opFirst\_input.txt)给出是否符合语法规则的判断，第一个例子(i+i)\*i/(i\*i/i)/#，不符合语法规则，给出结果如下，



第二个例子i+i\*i\*i+i/i\*i#，符合语法规则，



# 源代码附录

下面给出生成firstvt、lastvt、算符优先矩阵和算符优先分析的源代码。

首先是生成firstvt集,

void genFirstvt() {//生成firstvt集

    bool change = true;//作为firstvt集是否还增大的标志

    while (change) {

        change = false;

        for (NODES::iterator it = grammar.begin(); it != grammar.end(); ++it) {

            char left = it->first;//产生式左部

            for (set<string>::iterator vit = it->second.begin(); vit != it->second.end(); ++vit) {

                string right = \*vit;//产生式右部

                int len = right.size();//右部长度

                if (Terminate.find(right[0]) != Terminate.end()) {//右部首字符属于终结符号

                    if (firstvt[left].find(right[0]) == firstvt[left].end()) {//不在才加入

                        firstvt[left].insert(right[0]);

                        change = true;

                    }

                }

                else if (len >= 2 && Terminate.find(right[1]) != Terminate.end()) {//右部首字符属于非终结符号，第二个属于终结符号

                    if (firstvt[left].find(right[1]) == firstvt[left].end()) {//不在才加入

                        firstvt[left].insert(right[1]);

                        change = true;

                    }

                }

                else{//右部首字符属于非终结符号，第二个也属于非终结符号

                    for (set<char>::iterator iter = firstvt[right[0]].begin(); iter != firstvt[right[0]].end(); ++iter) {

                        if (firstvt[left].find(\*iter) == firstvt[left].end()) {//不在left的firstvt集

                            firstvt[left].insert(\*iter);

                            change = true;

                        }

                    }

                }

            }

        }

    }

}

生成lastvt集，

void genLastvt() {//生成lastvt集

    bool change = true;

    while (change) {

        change = false;

        for (NODES::iterator it = grammar.begin(); it != grammar.end(); ++it) {

            char left = it->first;//产生式左部

            for (set<string>::iterator vit = it->second.begin(); vit != it->second.end(); ++vit) {

                string right = \*vit;//产生式右部

                int len = right.size();//右部长度

                if (Terminate.find(right[len - 1]) != Terminate.end()) {//U -> ...a

                    if (lastvt[left].find(right[len - 1]) == lastvt[left].end()) {

                        lastvt[left].insert(right[len - 1]);

                        change = true;

                    }

                }

                else if ((len >= 2 && nonTerminate.find(right[len-2]) != nonTerminate.end()) || (len == 1 && nonTerminate.find(right[len - 1]) != nonTerminate.end())) {//U -> ...Q

                    for (set<char>::iterator iter = lastvt[right[len - 1]].begin(); iter != lastvt[right[len - 1]].end(); ++iter) {

                        if (lastvt[left].find(\*iter) == lastvt[left].end()) {

                            lastvt[left].insert(\*iter);

                            change = true;

                        }

                    }

                }

                else if(len >= 2){

                    if (lastvt[left].find(right[len - 2]) == lastvt[left].end()) {//U -> ...aQ

                        lastvt[left].insert(right[len - 2]);

                        change = true;

                    }

                }

            }

        }

    }

}

生成算符优先矩阵，

void genOpFirstMatrix() {//生成算符优先矩阵

    //首先处理#，# < firstvt(E), lastvt(E) > #

    for (set<char>::iterator it = firstvt['E'].begin(); it != firstvt['E'].end(); ++it) {

        Query q;

        q.end1 = '#';

        q.end2 = \*it;

        matrix[q] = -1;

    }

    for (set<char>::iterator it = lastvt['E'].begin(); it != lastvt['E'].end(); ++it) {

        Query q;

        q.end1 = \*it;

        q.end2 = '#';

        matrix[q] = 1;

    }

    Query q = { '#','#' };

    matrix[q] = 0;

    for (NODES::iterator it = grammar.begin(); it != grammar.end(); ++it) {

        char left = it->first;

        for (set<string>::iterator vit = it->second.begin(); vit != it->second.end(); ++vit) {

            string right = \*vit;//产生式右部

            int len = right.size();//右部长度

            for (int i = 0; i < len; ++i) {

                if (Terminate.find(right[i]) != Terminate.end()) {

                    if (i + 1 < len && (Terminate.find(right[i + 1]) != Terminate.end())) {//U -> ...ab...

                        Query q = { right[i],right[i + 1] };

                        matrix[q] = 0;

                    }

                    else if (i + 1 < len && (nonTerminate.find(right[i + 1]) != nonTerminate.end())) {//U -> ...aA...

                        for (set<char>::iterator iter = firstvt[right[i + 1]].begin(); iter != firstvt[right[i + 1]].end(); ++iter) {

                            Query q = { right[i],\*iter };

                            matrix[q] = -1;

                        }

                        if (i + 2 < len && (Terminate.find(right[i + 2]) != Terminate.end())) {//U -> ...aAb...

                            Query q = { right[i],right[i+2] };

                            matrix[q] = 0;

                        }

                    }

                    if (i > 0 && (nonTerminate.find(right[i - 1]) != nonTerminate.end())) {//U -> ...Ab...

                        for (set<char>::iterator it = lastvt[right[i-1]].begin(); it != lastvt[right[i-1]].end(); ++it) {

                            Query q;

                            q.end1 = \*it;

                            q.end2 = right[i];

                            matrix[q] = 1;

                        }

                    }

                }

            }

        }

    }

}

算符优先分析，

//传入中间文件，判断语句是否合法

bool opFirst(FILE\* fp) {

    opFirst\_stack.push\_back('#');

    char topEnd = '#';//当前栈中最靠前的终结符号(归约串的最右边的终结符号)

    int topEndPos = 0;

    char inputCh;//输入符号

    char reduceCh;//归约符号

    current = getCurrent(fp);

    while (!feof(fp)) {

        if (current.id == -1)

            break;

        if (current.id == ID) {

            inputCh = current.word[0];

            if (inputCh != 'i') {

                cout << "出现未定义标识符!" << endl;

                return false;

            }

        }

        else inputCh = convertion[current.id][0];

        Query q = { topEnd,inputCh };

        if (matrix.find(q) == matrix.end())

            return false;

        int priorRes = matrix[q];//优先级比较结果

        if (priorRes <= 0) {//topEnd 的优先级不大于输入符号的优先级

            opFirst\_stack.push\_back(inputCh);//将输入符号入栈

            topEnd = inputCh;//更新topEnd

            topEndPos = opFirst\_stack.size() - 1;

            current = getCurrent(fp);//输入串后移一位

        }

        else {//topEnd 的优先级大于输入符号的优先级

            //首先确定归约串

            while (matrix[{topEnd, inputCh}] == 1) {

                if (topEnd == opFirst\_stack.back()) {//归约串就是topEnd

                    string toBeReduced = "";

                    toBeReduced += topEnd;

                    reduceCh = reduce(toBeReduced);//归约符号

                    if (reduceCh == '0')

                        return false;

                    //更新topEnd及topEndPos

                    topEndPos--;

                    while (topEndPos >= 0 && Terminate.find(opFirst\_stack[topEndPos]) == Terminate.end())

                        topEndPos--;

                    if (topEndPos == -1)

                        topEndPos = 0;

                    topEnd = opFirst\_stack[topEndPos];

                    opFirst\_stack.pop\_back();//去除被归约的符号

                    opFirst\_stack.push\_back(reduceCh);//推入归约符号

                }

                else {

                    int nextPos = topEndPos;

                    for (int i = topEndPos - 1; i >= 0; i--)

                        if (i == 0 || (Terminate.find(opFirst\_stack[i]) != Terminate.end() && matrix[{opFirst\_stack[i], opFirst\_stack[nextPos]}] == -1)) {

                            string toBeReduced = "";//待归约串

                            int len = opFirst\_stack.size();

                            for (int k = i + 1; k < len; ++k)

                                toBeReduced += opFirst\_stack[k];

                            reduceCh = reduce(toBeReduced);

                            if (reduceCh == '0')

                                return false;

                            //更新topEnd

                            topEndPos = i;

                            topEnd = opFirst\_stack[topEndPos];

                            for (int t = len - 1; t != i; t--)//删除被归约串

                                opFirst\_stack.pop\_back();

                            opFirst\_stack.push\_back(reduceCh);

                            break;

                        }

                        else if(Terminate.find(opFirst\_stack[i]) != Terminate.end() ){

                            nextPos = i;

                        }

                }

            }

            opFirst\_stack.push\_back(inputCh);

            topEnd = inputCh;//更新topEnd

            topEndPos = opFirst\_stack.size() - 1;

            if (current.id == EOF)

                break;

            current = getCurrent(fp);//输入串后移一位

        }

    }

    return true;

}