编译原理大作业——算符优先分析表构造

程博元 安泰经济与管理学院 学号：517030910234

一、基本要求

从文本文件中读入一个上下文无关的算符文法，构造算符优先分析表并以文本文件的形式输出。

算符文法的格式要求：

（1）每个符号（终结符或者非终结符）可以是任意多个字符的组合，如 AA 可以作为一个非终结符，++可以作为一个终结符。

（2）两个符号之间有空格隔开，->与符号之间也由空格隔开，如 S -> (A)。

（3）文法中不出现|，如果存在 A -> a | b，则写为 A -> a 和 A -> b。

注：（1）本次设计中，设置了文法行数不超过 50 行（列表初始化为 50 个单位）。如果测试案例中出现了多于 50 行的文法，请联系 boyuancheng@sjtu.edu.cn 进行修改，谢谢老师。

（2）输入和输出文件在 compiler_theory 文件中。

二、设计原理

1、算符文法的定义

对于一个上下文无关文法 G，如果它是不含 ε – 产生式的文法，且产生式中不含两个相邻的非终结符，那么它是一个算符文法。

2、算符文法的性质

假定 G 是不含 ε – 产生式的算符文法。对于任何一对终结符 a、b，我们说：

（1）a 等于 b 当且仅当文法 G 中含有形如 P→ ・・・ab・・・或 P→・・・aQb・・・的产生式；

（2） a 小于 b 当且仅当 G 中含有形如 P→・・・aR・・・的产生式，而 R(+=>)b・・・或 R(+=>)Qb・・・；

（3） a 大于 b 当且仅当 G 中含有形如 P→・・・Rb・・・的产生式，而 R(+=>)・・・a 或 R(+=>)・・・aQ；

3、FIRSTVT 和 LASTVT 集合的定义

$$FIRSTVT(P)=\{\alpha \mid P\overset{+}{\Rightarrow}a\cdots \vee P\overset{+}{\Rightarrow}Qa\cdots, a\in V_T, Q\in V_N\}$$

$$LASTVT(P)=\{\alpha \mid P\overset{+}{\Rightarrow}\cdots a \vee P\overset{+}{\Rightarrow}\cdots aQ, a\in V_T, Q\in V_N\}$$

4、FIRSTVT 和 LASTVT 集合的求法

（1）若有产生式 $P \to a\cdots$ 或者 $P \to Qa\cdots$，则 $a \in FIRSTVT(P)$；

（2）若有产生式 $P \to Q\cdots$，则 $FIRSTVT(Q) \subseteq FIRSTVT(P)$。

（3）若有产生式 $P \to \cdots a$ 或者 $P \to \cdots aQ$，则 $a \in LASTVT(P)$；

（4）若有产生式 $P \to \cdots Q$，则 $LASTVT(Q) \subseteq LASTVT(P)$。

5、算符文法生成方法

（1）等于关系的求法参照上述定义；

（2）若产生式右部有 aP 的形式，则对每个 $b \in FIRSTVT(P)$，都有 $a < b$；

（3）若产生式右部有 Pb 的形式，则对每个 $a \in LASTVT(P)$，都有 $a > b$。

三、设计算法

本次设计中总共分为四部分，分别为终结符号和非终结符号的获取、FIRSTVT 集合和 LASTVT 集合的构造、优先关系表的生成、优先关系表的输出。

1、终结符号和非终结符号的获取

先对于文法进行遍历，获取产生式左部的所有符号即位非终结符号；然后对文法进行二次遍历，获取所有符号，并将非终结符号过滤，剩余的即位终结符号。

2、FIRSTVT 集合和 LASTVT 集合的构造

构造部分的核心是两个判断矩阵 FIRSTVT 和 LASTVT，FIRSTVT 格式如下

	终结符 1	终结符 2	终结符 3	终结符 4
非终结符 1	FIRSTVT[0][0]	FIRSTVT[0][1]	FIRSTVT[0][2]	FIRSTVT[0][3]
非终结符 2	FIRSTVT[1][0]	FIRSTVT[1][1]	FIRSTVT[1][2]	FIRSTVT[1][3]

LASTVT 格式如下：

	终结符 1	终结符 2	终结符 3	终结符 4
非终结符 1	LASTVT[0][0]	LASTVT[0][1]	LASTVT[0][2]	LASTVT[0][3]
非终结符 2	LASTVT[1][0]	LASTVT[1][1]	LASTVT[1][2]	LASTVT[1][3]

同时，设立了两个栈结构 vn、vt，放置成对出现的（非终结符，终结符）。

FIRSTVT 集合的构造：

（1）算法第一步：若有产生式 $P \to a\cdots$ 或者 $P \to Qa\cdots$，则 $a \in FIRSTVT(P)$

对于每一条产生式$P \to \cdots$，寻找开头的终结符号a，或者紧接在开头非终结符号后面的终结符号a，将FIRSTVT对应的位置（即P，a所在位置）置为true，同时把(P,a)入栈。

（2）算法第二步：若有产生式$P \to Q\cdots$，则$FIRSTVT(Q) \subseteq FIRSTVT(P)$

出栈一个符号对(Q,a)，然后寻找是否存在$P \to Q\cdots(P \neq Q)$的形式，如果存在，将FIRSTVT对应的位置（即P，a所在位置）置为true，同时把(P,a)入栈。

通过以上两个步骤可以成功构造FIRSTVT集合。

LASTVT集合的构造：

（1）算法第一步：若有产生式$P \to \cdots a$或者$P \to \cdots aQ$，则$a \in LASTVT(P)$

对于每一条产生式$P \to \cdots$，寻找结尾的终结符号a，或者紧接在结尾非终结符号前面的终结符号a，将LASTVT对应的位置（即P，a所在位置）置为true，同时把(P,a)入栈。

（2）算法第二步：若有产生式$P \to \cdots Q$，则$LASTVT(Q) \subseteq LASTVT(P)$

出栈一个符号对(Q,a)，然后寻找是否存在$P \to \cdots Q(P \neq Q)$的形式，如果存在，将LASTVT对应的位置（即P，a所在位置）置为true，同时把(P,a)入栈。

通过以上两个步骤可以成功构造LASTVT集合。

3、优先关系表的生成

这一部分的关键是table表，其格式为：

	终结符号1	终结符号2	终结符号3	终结符号4
终结符号1	table[0][0]	table[0][1]	table[0][2]	table[0][3]
终结符号2	table[1][0]	table[1][1]	table[1][2]	table[1][3]
终结符号3	table[2][0]	table[2][1]	table[2][2]	table[2][3]
终结符号4	table[3][0]	table[3][1]	table[3][2]	table[3][3]

每个位置表示了两个终结符号的关系，行为左部，列为右部。2代表无关，0代表等于，-1代表小于，1代表大于。

优先关系生成部分处理每一行文法，识别右部每一个符号：

（1）如果为终结符号：

如果下一位为终结符号，则置为=；若为非终结符号，则获取其所有FIRSTVT元素并置为<；

若非终结符号后面仍有终结符号，则两个终结符号置为=。

（2）如果为非终结符号：

如果下一位是终结符号，则获取其所有 LASTVT 元素并置为>。

4、优先关系表的输出

输出的时候，将$与各个终结符号的关系表示。如果 $a \in FIRSTVT(E)$，则 $\$ < a$；如果 $a \in LASTVT(E)$，则 $a > \$$。其中 E 为文法的开始符号。

其余按照 3 中生成的 table 输出即可。

四、设计方案

1、终结符号和非终结符号的获取

```
vector<string> Vt;  //终结符向量
    vector<string> Vn; //非终结符向量
    vector<string> v;
    vector<string> grammar[50];  //将文法的所有信息保存，文法最多有 20 行
    ifstream in("输入.txt");
    string s;  //读入的一行字符串
    int n = 0, grammar_len = 0;
    bool judge;

    //获取非终结符向量
    while (getline(in, s))
    {
        v = split(s);
        Vn.push_back(v[0]);   //第一个字符是非终结符
        grammar_len++;  //记录文法的总行数
    }
    in.close();

    //获取终结符向量
    in.open("输入.txt");
    while (getline(in, s))
    {
        v = split(s);
        grammar[n] = v;
        n++;
        for (int i = 2; i != v.size(); ++i) {
            judge = false;
            for (int j = 0; j != Vn.size(); ++j) {
```

```
                if (Vn[j] == v[i])  judge = true;   //检验该符号是否在非终结符向量中
            }
            if (judge != true)  Vt.push_back(v[i]);  //放入终结符向量中
        }
    }
    in.close();

    //去除 Vn 和 Vt 中相同的元素
    Vn.erase(unique(Vn.begin(), Vn.end()), Vn.end());
    Vt.erase(unique(Vt.begin(), Vt.end()), Vt.end());
```

2、FIRSTVT 集合和 LASTVT 集合的构造

```
//获取 FIRSTVT 集合
bool **  get_FIRSTVT(vector<string>Vn, vector<string>Vt,vector<string>* grammar,int grammar_len) {
    cout << "_______获取 FIRSTVT_______" << endl;
    string s1, s2, s3;
    int index1, index2, index3;
    bool** FIRSTVT ;   //判断矩阵，每一行是非终结符，列是终结符
    FIRSTVT = (bool **)malloc(50 * sizeof(bool*));
    for (int i = 0; i != 50; ++i) {
        FIRSTVT[i] = (bool*)malloc(50 * sizeof(bool));
    }
    //建立两个栈，两个栈联合使用
    stack<string> vn;
    stack<string> vt;
    //判断矩阵所有元素置为假
    for (int i = 0; i != 50; ++i) {
        for (int j = 0; j != 50; ++j) {
            FIRSTVT[i][j] = false;
        }
    }
    //执行算法第一步
    for (int i = 0; i != grammar_len; ++i) {  //处理文法的每一行
        s1 = grammar[i][0];  //当前正在处理的非终结符
        index1 = find(Vn, s1);
        index2 = find(Vt, grammar[i][2]);  //判断该行第二个字符是否是终结符
        if (index2 != -1) {  //是终结符
            FIRSTVT[index1][index2] = true;
            vn.push(s1);
            vt.push(grammar[i][2]);  //将符号对入栈
            cout << s1 << "和" << grammar[i][2] << "入栈" << endl;
        }
```

```
        else {  //不是终结符
            if (grammar[i].size() > 3) {  //先判断该行文法是否超过 3 个字符
                index3 = find(Vt, grammar[i][3]);
                if (index3 != -1) {  //非终结符的下一个元素是终结符
                    FIRSTVT[index1][index3] = true;
                    vn.push(s1);
                    vt.push(grammar[i][3]);  //将符号对入栈
                    cout << s1 << "和" << grammar[i][3] << "入栈" << endl;
                }
            }
        }
    }
    //执行算法第二步
    while (!vn.empty()) {  //只要栈非空，就持续循环
        s1 = vn.top();  //pop 一个非终结符号
        s2 = vt.top();  //pop 一个终结符号
        vn.pop();
        vt.pop();
        for (int k = 0; k != grammar_len; k++) { //在所有产生式中，寻找是否有某个产生式
的->第一个符号是 s1
            if (grammar[k][2] == s1) {
                if (grammar[k][0] != s1) {   //前提是非终结符和 s1 是不同的
                    index1 = find(Vn, grammar[k][0]);  //寻找当前非终结符所在位置
                    index2 = find(Vt, s2);  //获得该终结符所在位置
                    FIRSTVT[index1][index2] = true;
                    vn.push(grammar[k][0]);
                    vt.push(s2);  //将符号对入栈
                    cout << grammar[k][0] << "和" << s2 << "入栈" << endl;
                }
            }
        }
    }
    return FIRSTVT;
}


//获取 LASTVT 集合
bool **  get_LASTVT(vector<string>Vn, vector<string>Vt, vector<string>* grammar, int
grammar_len) {
    cout << "_______获取 LASTVT_______" << endl;
    string s1, s2, s3;
    int index1, index2, index3;
    bool** LASTVT;   //判断矩阵，每一行是非终结符，列是终结符
    LASTVT = (bool **)malloc(50 * sizeof(bool*));
```

```
    for (int i = 0; i != 50; ++i) {
        LASTVT[i] = (bool*)malloc(50 * sizeof(bool));
    }
    //建立两个栈，两个栈联合使用
    stack<string> vn;
    stack<string> vt;
    //判断矩阵所有元素置为假
    for (int i = 0; i != 50; ++i) {
        for (int j = 0; j != 50; ++j) {
            LASTVT[i][j] = false;
        }
    }
    //执行算法第一步
    for (int i = 0; i != grammar_len; ++i) {  //处理文法的每一行
        s1 = grammar[i][0];  //当前正在处理的非终结符
        index1 = find(Vn, s1);
        index2 = find(Vt, grammar[i][grammar[i].size() - 1]);  //判断该行最后一个字符是否是终结符
        if (index2 != -1) {  //是终结符
            LASTVT[index1][index2] = true;
            vn.push(s1);
            vt.push(grammar[i][grammar[i].size() - 1]);  //将符号对入栈
            cout << s1 << "和" << grammar[i][grammar[i].size() - 1] << "入栈" << endl;
        }
        else {  //不是终结符
            if (grammar[i].size() > 3) {  //先判断该行文法是否超过 3 个字符
                index3 = find(Vt, grammar[i][grammar[i].size() - 2]);
                if (index3 != -1) {  //非终结符的下一个元素是终结符
                    LASTVT[index1][index3] = true;
                    vn.push(s1);
                    vt.push(grammar[i][grammar[i].size() - 2]);  //将符号对入栈
                    cout << s1 << "和" << grammar[i][grammar[i].size() - 2] << "入栈" << endl;
                }
            }
        }
    }
    //执行算法第二步
    while (!vn.empty()) {  //只要栈非空，就持续循环
        s1 = vn.top();  //pop 一个非终结符号
        s2 = vt.top();  //pop 一个终结符号
        vn.pop();
        vt.pop();
        for (int k = 0; k != grammar_len; k++) { //在所有产生式中，寻找是否有某个产生式
```

```
的->第一个符号是 s1
            if (grammar[k][grammar[k].size() - 1] == s1) {
                if (grammar[k][0] != s1) {   //前提是非终结符和 s1 是不同的
                    index1 = find(Vn, grammar[k][0]);  //寻找当前非终结符所在位置
                    index2 = find(Vt, s2);  //获得该终结符所在位置
                    LASTVT[index1][index2] = true;
                    vn.push(grammar[k][0]);
                    vt.push(s2);  //将符号对入栈
                    cout << grammar[k][0] << "和" << s2 << "入栈" << endl;
                }
            }
        }
    }
    return LASTVT;
}

//输出 FIRSTVT 和 LASTVT
void make_judge_table(bool** judge_table, vector<string>Vn, vector<string>Vt, string name)
{
    cout << "_______"<<name<<"_______" << endl;
    for (int i = 0; i != Vn.size(); ++i) {
        cout << Vn[i] << " : ";
        for (int j = 0; j != Vt.size(); ++j) {
            if (judge_table[i][j] == true) {
                cout << " " << Vt[j] << " ";
            }
        }
        cout << endl;
    }
    cout << endl;
}
```

3、优先关系表的生成

```
//构造算符优先表
int ** make_table(vector<string>Vn, vector<string>Vt, vector<string>* grammar, int
grammar_len, bool** FIRSTVT, bool** LASTVT)
{
    string s1, s2, s3;
    int index1, index2, index3, index4;

    int** table;   //算符优先表，行列均为终结符号
    //算符优先表初始化
    table = (int **)malloc(50 * sizeof(int*));
```

```
for (int i = 0; i != 50; ++i) {
    table[i] = (int*)malloc(50 * sizeof(int));
}
//将每个元素设为2，表示当前两者没有关系
for (int i = 0; i != 50; ++i) {
    for (int j = 0; j != 50; ++j) {
        table[i][j] = 2;
    }
}

for (int i = 0; i != grammar_len; ++i) {   //处理文法的每一行
    for (int j = 2; j != grammar[i].size(); ++j) {  //处理文法从->后面的每一个符号
        s1 = grammar[i][j];
        index1 = find(Vt, s1);
        //如果s1是终结符号
        if ((index1 != -1) && (j != grammar[i].size() - 1)) {
            s2 = grammar[i][j + 1];
            index2 = find(Vt, s2);
            if (index2 != -1) {  //如果s1的下一位也是终结符号
                table[index1][index2] = 0;
            }
            else {  //如果s1的下一位是非终结符号
                index3 = find(Vn, s2);
                for (int k = 0; k != 50; ++k) {   //获取FIRSTVT(s2)的所有元素
                    if (FIRSTVT[index3][k] == true) {
                        table[index1][k] = -1;
                    }
                }
                if (j != grammar[i].size() - 2) {  //如果后面仍然有终结符号
                    s3 = grammar[i][j + 2];
                    index4 = find(Vt, s3);
                    if (index4 != -1) {
                        table[index1][index4] = 0;
                    }
                }

            }
        }

        //如果s1是非终结符号
        if ((index1 == -1) && (j != grammar[i].size() - 1)) {
            s2 = grammar[i][j + 1];
            index2 = find(Vt, s2);
            if (index2 != -1) {  //下一位是终结符号
```

```
                    index3 = find(Vn, s1);
                    for (int k = 0; k != 50; ++k) {    //获取 LASTVT(s1)的所有元素
                        if (LASTVT[index3][k] == true) {
                            table[k][index2] = 1;
                        }
                    }
                }
            }
        }
    }
    return table;
}
```

4、优先关系表的输出

```
//将算符优先表输出
void print_table(int ** table, vector<string>Vn, vector<string>Vt, vector<string>* grammar,
bool** FIRSTVT, bool** LASTVT)
{
    ofstream out("输出.txt");
    string s = grammar[0][0];   //获取开始符号
    int index;

    cout << "_____算符优先表_____" << endl;
    out << "_____算符优先表_____" << endl;
    cout << "\t";
    out << "\t";
    //输出表格第一行
    for (int i = 0; i != Vt.size(); ++i) {
        cout << Vt[i] << "\t";
        out << Vt[i] << "\t";
    }
    cout << "$\t" << endl;  //将$补足
    out << "$\t" << endl;

    //输出表格剩余部分
    for (int i = 0; i != Vt.size(); ++i) {
        //输出行头（即不等式左边的终结符）
        cout << Vt[i] << "\t";
        out << Vt[i] << "\t";
        //输出其和各个终结符之间的优先关系
        for (int j = 0; j != Vt.size(); ++j) {
            if (table[i][j] == 2) {  //若为 2，说明两者无关
                cout << "\t";
```

```
                out << "\t";
            }
            if (table[i][j] == 0) { //若为0，则为=
                cout << "=" << "\t";
                out << "=" << "\t";
            }
            if (table[i][j] == -1) { //若为-1，则为<
                cout << "<" << "\t";
                out << "<" << "\t";
            }
            if (table[i][j] == 1) { //若为1，则为>
                cout << ">" << "\t";
                out << ">" << "\t";
            }
        }
        //判断当前处理的终结符和$的关系
        index = find(Vn, s);
        if (LASTVT[index][i] == true) {
            cout << ">\t";
            out << ">\t";
        }
        cout << endl;
        out << endl;
    }

    //输出表格最后一行，即$的部分
    cout << "$\t";
    out << "$\t";
    //判断$和每一个非终结符之间的关系
    index = find(Vn, s);
    for (int j = 0; j != Vt.size(); ++j) {
        if (FIRSTVT[index][j] == true) {
            cout << "<\t";
            out << "<\t";
        }
        else {
            cout << "\t";
            out << "\t";
        }
    }
    cout << "=\t";
    out << "=\t"; //$和$之间为=关系
}
```

5、辅助函数

```
//判断某个元素是否在一个 vector 中，如果存在返回索引，否则返回-1
int find(vector<string> v, string str) {
    for (int j = 0; j != v.size(); ++j) {
        if (v[j] == str)  return j;
    }
    return -1;
}

// 将读入的字符串按照空格进行分割
vector<string> split(const string str) {
    vector<string> str_split;
    const string split_token = " \n";
    int i = 0;  //标记位置，将输入的字符串从前到后依次扫描

    while (i != str.size()) {
        //找到字符串中首个有效字母；
        int flag = 0;
        while (i != str.size() && flag == 0) {
            for (int x = 0; x < 2; ++x)
                if (str[i] == split_token[x]) {
                    ++i;
                    flag = 0;
                }
                else { flag = 1; }
        }
        //将两个空格之间的字符串取出；
        flag = 0;
        int j = i;  //i 代表字符串的起点，j 代表字符串的终点
        while (j != str.size() && flag == 0) {
            for (int x = 0; x < 2; ++x)
                if (str[j] == split_token[x]) {
                    flag = 1;
                }
            if (flag == 0)
                ++j;
        }
        if (i != j) {
            str_split.push_back(str.substr(i, j - i));  //将一段字符串取出
            i = j;
        }
    }

    return str_split;
```

```
}
```

6、主函数

```
int main()
{
    vector<string> Vt;  //终结符向量
    vector<string> Vn; //非终结符向量
    vector<string> v;
    vector<string> grammar[50];  //将文法的所有信息保存，文法最多有 20 行
    ifstream in("输入.txt");
    string s;  //读入的一行字符串
    int n = 0, grammar_len = 0;
    bool judge;

    //获取非终结符向量
    while (getline(in, s))
    {
        v = split(s);
        Vn.push_back(v[0]);   //第一个字符是非终结符
        grammar_len++;  //记录文法的总行数
    }
    in.close();

    //获取终结符向量
    in.open("输入.txt");
    while (getline(in, s))
    {
        v = split(s);
        grammar[n] = v;
        n++;
        for (int i = 2; i != v.size(); ++i) {
            judge = false;
            for (int j = 0; j != Vn.size(); ++j) {
                if (Vn[j] == v[i])  judge = true;   //检验该符号是否在非终结符向量中
            }
            if (judge != true)  Vt.push_back(v[i]);  //放入终结符向量中
        }
    }
    in.close();

    //去除 Vn 和 Vt 中相同的元素
    Vn.erase(unique(Vn.begin(), Vn.end()), Vn.end());
    Vt.erase(unique(Vt.begin(), Vt.end()), Vt.end());
```

```
    //获取 FIRSTVT 集合
    bool ** FIRSTVT;
    FIRSTVT = get_FIRSTVT(Vn, Vt, grammar, grammar_len);

    //获取 LASTVT 集合
    bool ** LASTVT;
    LASTVT = get_LASTVT(Vn, Vt, grammar, grammar_len);

    //输出 FIRSTVT 集合
    make_judge_table(FIRSTVT, Vn, Vt, "FIRSTVT");

    //输出 FIRSTVT 集合
    make_judge_table(LASTVT, Vn, Vt, "LASTVT");

    //获取算符优先表
    int **table;
    table = make_table(Vn, Vt, grammar, grammar_len, FIRSTVT, LASTVT);

    //将算符优先表输出
    print_table(table, Vn, Vt, grammar, FIRSTVT, LASTVT);
}
```

五、工程测试

1、测试一

文法：

S -> a

S -> b

S -> c

S -> e

S -> f

S -> g

S -> h

S -> i

S -> j

S -> k

S -> l

S -> m

S -> n

S -> o

S -> p

S -> q

S -> r

S -> aa

S -> (A)

A -> S d A

A -> S

测试结果：

算符优先表

	a	b	c	e	f	g	h	i	j	k	l	m	aa	(	)	d	$
a															>	>	>
b															>	>	>
c															>	>	>
e															>	>	>
f															>	>	>
g															>	>	>
h															>	>	>
i															>	>	>
j															>	>	>
k															>	>	>
l															>	>	>
m															>	>	>
aa															>	>	>
(	<	<	<	<	<	<	<	<	<	<	<	<	<	<	=	<	
)															>	>	>
d	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	
$	<	<	<	<	<	<	<	<	<	<	<	<	<	<			=

2、测试二

文法：

E -> E + T

E -> T

T -> T * F

T -> F

F -> (E)

F -> id

测试结果：

算符优先表

	+	*	(	)	id	$
+	>	<	<	>	<	>
*	>	>	<	>	<	>
(	<	<	<	=	<	
)	>	>		>		>
id	>	>		>		>
$	<	<	<		<	=

3、测试三

文法：

E -> E + T

E -> T

T -> T * F

T -> F

F -> P ~ F

F -> P

P -> (E)

P -> id

测试结果：

算符优先表

	+	*	~	(	)	id	$
+	>	<	<	<	>	<	>
*	>	>	<	<	>	<	>
~	>	>	<	<	>	<	>
(	<	<	<	<	=	<	
)	>	>	>		>		>
id	>	>	>		>		>
$	<	<	<	<		<	=

六、总结

本次大作业中，我进行了算符优先分析分析表的构建，对语法分析部分有了更加深入的了解。同时，在调试过程中出现了一些问题，如初始化错误、未解决循环递归等问题，也通过不断修复获得了解决，大大提高了我程序分析和算法实现的能力。作为社科学院的 CS 爱好者，这次作业让我初步窥见了计算机底层的设计模式，对编译器有了更加深刻的认知。

最后，感谢张冬茉老师对《编译原理》课程的认真教授，祝老师身体健康，工作顺利！