编译原理实验报告

王子龙 18281218 wangzilong@bjtu.edu.cn

实验环境

类目	详情
操作系统	macOS Big Sur 11.4
CPU	Intel Core i5-7260U@2.3Ghz x2
IDE	CLion 2021.1.2 Build #CL-211.7442.42
Compiler	Apple clang version 11.0.0 (clang-1100.0.33.8)

实验要求（实验功能描述）

实验项目

完成以下描述赋值语句SLR(1)文法语法制导生成中间代码四元式的过程。

$$
\begin{aligned}
G[S]: & E \rightarrow E+T|E-T|T \\
& T \rightarrow T*F|T/F|F \\
& F \rightarrow (E)|i
\end{aligned}
$$

设计说明

终结符号`i`为用户定义的简单变量，即标识符的定义。

设计要求

- 构造文法的SLR(1)分析表，设计语法制导翻译过程，给出每一产生式对应的语义动作；
- 设计中间代码四元式的结构；
- 输入串应是词法分析的输出二元式序列，即某赋值语句“专题1”的输出结果，输出为赋值语句的四元式序列中间文件；
- 设计两个测试用例（尽可能完备），并给出程序执行结果四元式序列。

任务分析

重点解决SLR(1)分析表构造，赋值语句文法的改写和语义动作的添加。

主要数据结构描述

文法结构体`GLO`

```
typedef struct Glo
{
    VN *VNs;
    VN *VTs;
} GLO;
```

其中`VNs`为非终结符号数组，`VTs`为终结符号数组，定义`VN`类型如下:

符号数组

```
typedef struct vn
{
    string Key;
    int Seq;
    int *Funcs;
    int *first; //以-1为结束标志
    int *follow; //以-1为结束标志
} VN;
```

每个`VN`类型变量包含一个`String`类型的`Key`，存储其具体名称，一个编号`Seq`，非终结符号还有对应的产生式指针`Func`，其中存储对应产生式的编号。以及`First`集和`Follow`集指针，其中存储的为终结符号的编号。

本实验中的`VNs`和`VTs`定义如下（其中“0”表示空产生式）：

```
VN vns[] = {{"E", 1}, {"T", 2}, {"F", 3}};
VN vts[] = {{"+", 4}, {"-", 5}, {"*", 6}, {"/", 7}, {"(", 8}, {")", 9}, {"i", 10}, {"#", 11}};
```

每个产生式右部的数据结构定义如下：

```
//每个导出式右部的字符串,以及对应的符号序号将在initial中链接， “0”为空导出式
FUNC func[] = {{"E+T"}, {"E-T"}, {"T"}, {"T*F"}, {"T/F"}, {"F"}, {"(E)"}, {"i"}, {"0"}};

//每个导出式右部的序号数组,以0为一个右部的结束标志,如TE' 为3, 2, 0, 里面的数字对应vns和vnt中的序号
int func_seq[][10] = {{1, 4, 2, 0}, {1, 5, 2, 0}, {2, 0}, {2, 6, 3, 0}, {2, 7, 3, 0}, {3, 0}, {8, 1, 9, 0}, {10, 0},
{11, 0}};
```

因为一个非终结符号可以导出多个产生式右部，故要对每个产生式右部编号，让`VN`结构体中的`Funcs`数组存储每个非终结结符号对应的所有右部编号。按照 E,E'...的顺序，分别对应的产生式右部编号为(0为结束标志):

```
//每个非终结符号包含的所有导出式的序号， 里面的数字对应func_seq中的数组序号，如 1 对应func[0], 2 对应func[1]
int func_vn[][10] = {{1, 2, 3, 0}, {4, 5, 6, 0}, {7, 8, 0}};
```

状态

使用结构体`I`对DFA中的状态进行存储，其中`number`对应了状态变化，`vector<form> project`则对应了其状态族，用于存储该状态中的产生式。

```
typedef struct i
{
    int number; //状态编号;
    vector<form> project; //项目族集
    i(int num = 0) :number(num) {};
}I;
```

四元式

四元式的格式为$(op, arg1, arg2, res)$，因此使用一个结构体`Qua`对四元式进行存储。

```
typedef struct Qua //四元式, op, arg1, arg2, res
{
    string op;
    string arg1;
    string arg2;
    string res;
    Qua(string o = "", string a1 = "", string a2 = "", string r = "") :op(o), arg1(a1), arg2(a2), res(r) {};
}Quaternion;
```

使用结构体`binary`进行符号表的存储。在构造四元式的过程中，需要对于上一次的操作中新生成的结果进行记录，以便生成新的四元式时作为参数加入其中；故结构体中的`key`代表某一运算结果的键值；`string`类的`value`表示该变量对应的算术表达式（属性）。

```
struct binary
{
    int key;
    string val;
    binary(int x = 0, string s = "") :key(x), val(s) {};
};
```

使用结构体`act`对*ACTION*进行存储，其中tag代表该动作，*e.g.*r规约，S转移...，`action`表示转移后的状态或使用哪条产生式进行规约。`vn`代表规约时产生式的左部。

```
typedef struct act
{

    char tag; //r归约，  s转移
    int action;  //转移到哪个状态 或者 用哪条产生式归约
    int vn; //归约时产生式的左部
    act(char t = '0', int a = -1, int v = -1) : tag(t), action(a), vn(v) {};
}Act;
```

程序结构描述

构造DFA以及Action/GoTo的构造

SLR(1)分析表构造=LR(0)分析表+SLR(1)规则

$$C_i = \{U \to a \cdot b\beta,\ V \to a\cdot, W \to a\cdot\}$$

考察$FOLLOW(V)$，$FOLLOW(W)$及$\{b\}$，若它们两两不相交，则可采用下面的方法，对C_i中各个项目所对应的分析动作加以区分。

$$FOLLOW(V) = \{a | S'\# => *\ldots Va\ldots, a \in V_t \cup \{\#\}\}$$

对任何输入符号a:

- 当$a = b$时，置$ACTION[i, b] =$“移进”
- 当$a \in FOLLOW(V)$时，置$ACTION[i, a] = \{$按产生式$V \to \alpha\}$规约
- 当$a \in FOLLOW(W)$时，置$ACTION[i, a] = \{$按产生式$W \to \alpha\}$规约
- 当a不属于上述三种情况之一时，置$ACTION[i, a] =$” $ERROR$ ”

上述用来解决分析动作冲突的方法称为**SLR(1)规则**。

- 拓广文法G'
- 构造(对G')LR(0)有效项目集族C和Go函数
- 若$GO(C_i, a) = C_j a \in V_t$，置$ACTION(i, a) = S_j$
- 若$GO(C_i, A) = C_j a \in V_n$，置$GOTO(i, A) = j$
- 若$A \to \alpha\cdot \in C_j$，且$a \in FOLLOW(A)$，$a \in V_t$，置$ACTION(i, a) = r_j$（$A \to \alpha$为第$j$个产生式）
- 若$S \to \delta\cdot \in C_k$，（$S$为拓广文法开始符号），置$ACTION(k, \#) =$`acc`
- 其它：空白（*error*）

```
vector<I> build_DFA()
{
    vector<I> D;
    int curi = 0; //当前正在检查的项目族编号
    vector<int> check; //当前检查的项目族可以接受的字符编号，  比如I0 可以接受 S，  V ，i
    vector<form> reduction; //存储当前项目族中可以归约的产生式
    int pro_len;
    int had;
    //建立I0;
    I newi(0);
    form new_form(func[0], 1, 0, 0);
    vector<form> vec_form;
    vec_form.push_back(new_form);
    newi.project = closure(vec_form);
    //
    act_row new_actrow;
    goto_row new_gotorow;
    Action.push_back(new_actrow); //Action表新增一行
    GoTo.push_back(new_gotorow); //Goto表新增一行

    D.push_back(newi);// I0入DFA

    while (curi <= D.size() - 1) //当curi大于D.size()时，说明已经没有新的项目族集再产生了
    {
        pro_len = D[curi].project.size(); //DFA栈顶的项目族  产生式的数量
        for (int i = 0; i < pro_len; i++) //寻找对于当前I，能接受的字符集，  比如I0 可以接受 S，  V ，i
        {
```

```
            int tag = D[curi].project[i].tag; //查看当前I每个产生式的tag后面的字符，tag从0开始
            FUNC f = D[curi].project[i].func; //项目族集中的产生式
            if (f.Seqs[tag] != 0 && !vec_find(check, f.Seqs[tag])) //tag后面的符号不是 0（产生式结束标志），  也没有出现在check中
                check.push_back(f.Seqs[tag]); //可以接受的字符入check栈
            else if (f.Seqs[tag] == 0) //已经finished的产生式入归约栈
                reduction.push_back(D[curi].project[i]);
        }

        //检查当前项目族D[curi]导致归约的字符。
        while (!reduction.empty())
        {
            form top = reduction.back();
            reduction.pop_back();
            int vn = top.vn; //vn从1开始
            int vt_len = sizeof(vts) / sizeof(vts[0]);
            for (int i = 0; i < vt_len; i++)
            {
                if (follow_has(vns[vn - 1].follow, i + 7))  //可以归约的产生式左部的follow集 含有 当前vt
                {
                    Action[curi].row[i].tag = 'r';  //归约标志
                    Action[curi].row[i].vn = vn; //产生式左部编号
                    Action[curi].row[i].action = top.func.num; //使用的产生式的编号
                }
            }
        }

        //下面检查的都是当前项目族 D[curi] 移进的字符，不是可以归约的.
        while (!check.empty()) //依次检查check
        {
            int v = check.back();
            check.pop_back();
            vector<form> new_form;
            vector<form> closured;
            for (int i = 0; i < pro_len; i++) //遍历产生式，将可以接受check里面字符的产生式， tag往后移一位之后加入new_form
            {
                int tag = D[curi].project[i].tag; //tag从0开始
                int vn = D[curi].project[i].vn; //产生式的左部
                FUNC f = D[curi].project[i].func; //项目族集中的产生式
                if (f.Seqs[tag] == v) //可以接受 v  的产生式
                {
                    form temp(f, vn, tag + 1);
                    if (f.Seqs[tag + 1] == 0)//可以归约了， finished设置为1
                        temp.finish = 1;
                    else
                        temp.finish = 0;
                    new_form.push_back(temp);
                }
            }
            closured = closure(new_form); //求一步closure
            had = already_hadI(D, closured); //检查是否之前出现过相同的项目族
            if (had == -1) //不存在相同的，新建I，加入DFA
            {
                I newi(D.size());
                newi.project = closured;
                D.push_back(newi);
                //下面为更新Action和Goto表
                if (v_is_final(v)) //接受的VT， 更新Action表
                {
                    Action[curi].row[v - 7].tag = 's'; //s代表移进
                    Action[curi].row[v - 7].action = D.size() - 1; //转移到的状态编号
                }
                else  //接受的VN， 更新Goto表
                    GoTo[curi].row[v - 2] = D.size() - 1;  //转移到的状态编号， 要减2，因为Goto表中从S开始，不是S'
                //下面给新的I，在Action和Goto表中新增一行
                act_row new_actrow;
```

```
                goto_row new_gotorow;
                Action.push_back(new_actrow); //Action表新增一行
                GoTo.push_back(new_gotorow); //Goto表新增一行
            }
            else //存在相同的，在Action和Goto中设置状态转移
            {
                if (v_is_final(v)) //接受的VT， 更新Action表
                {
                    Action[curi].row[v - 7].tag = 's'; //s代表移进
                    Action[curi].row[v - 7].action = had; //转移到的状态编号
                }
                else  //接受的VN， 更新Goto表
                    GoTo[curi].row[v - 2] = had;  //转移到的状态编号， 要减2，因为Goto表中从S开始，不是S'
            }
        }
        curi++; //检查下一个项目族集。
    }
    return D;
}
```

SLR1分析

SLR()完成最终整个SLR(1)文法的分析任务，也包括分析过程中完成的四元式构建。在分析过程起始时，首先将状态I0压入栈中，分析开始：若ACTION(栈顶，当前符号)不是acc（在程序中为r0），则开始循环：如果是err（在程序中使用0代替），报错；否则如果是sj，状态栈、符号栈、属性栈均压栈，读取下一个符号；如果是ri，则通过需要规约第i条产生式，对于三个栈，执行第i条产生式右部的符号数次pop()操作，同时更新属性栈，用以后续的四元式生成工作；在生成四元式时，首先判断正在规约的产生式是否可以产生四元式，同时确定四元式的op，其次判断两个参数（arg1，arg2）是否为变量or变量组成的表达式。如果是表达式，则查找之前的符号表，进行代替操作。若中间不报错而跳出循环，则成功匹配。至此，按照事先约定好的输出格式进行输出，分析任务完成。

```
bool SLR1()
{
    vector<string> value; //存储符号的属性
    vector<int> analysis; //分析容器存储 字符编号
    vector<int> state;   //状态容器存储  状态编号
    analysis.push_back(15);
    state.push_back(0);
    value.push_back("#");
    //current为当前还未入栈，正在分析的字符，cur_state为当前状态编号,cur_act为对于分析字符的动作，r_vn为归约动作时的产生式左部符号，r_len为归约动作时产生式右部的长度
    int current, cur_state, cur_act, r_vn, r_len;
    binary next; //存储下一个二元组
    string cur_val; //如果当前是i，存储其val
    char tag;
    string a1, a2, r, o;
    bool is_r = false; //如果此轮分析出现了归约，那么current应该变成归约后的VN，也不应该再读新的输入字符串
    while (!(state.back() == 0 && analysis.back() == 1)) //还未Acc
    {
        print_ana(analysis);
        print_state(state);
        print_value(value);
        cout << endl;
        cur_state = state.back();
        if (!is_r)
        {
            next = read_next(); //读取下一个字符
            current = next.key;
            cur_val = next.val;
        }



        if (v_is_final(current))  //是VT，查Action表
        {
            cur_act = Action[cur_state].row[current - 7].action; //状态容器顶
```

```
            tag = Action[cur_state].row[current - 7].tag;  //是s还是r
            if (cur_act != -1 && tag == 's') //是s，  移进项目，action为状态转移编号
            {
                state.push_back(cur_act);
                analysis.push_back(current);
                value.push_back(cur_val); //属性值进容器
                is_r = false;
            }
            else if (cur_act != -1 && tag == 'r') //是r，  归约项目，action为产生式编号，  func中只有编号1,2,3,5,6可以产生四元式，且只有1不用产生新的临时变量
            {
                r_vn = Action[cur_state].row[current - 7].vn;
                r_len = func[cur_act].len;

                //下面为生成四元式，更改value内容
                if (had_qua(cur_act))//可以产生四元式
                {
                    if (cur_act == 1) //S->V=E
                    {
                        a1 = value.back();
                        value.pop_back();
                        value.pop_back(); // E和=的val出栈
                        r = value.back();
                        value.pop_back();
                        value.push_back(a1); //将新的value入栈
                        Quaternion newq("=", a1, "", r);
                        Q.push_back(newq);
                    }
                    else  //E->E+T, E->E-T, T->T*F, T->T/F
                    {
                        a2 = value.back();
                        value.pop_back();
                        o = value.back();
                        value.pop_back();
                        a1 = value.back();
                        value.pop_back();
                        r = newtemp();
                        value.push_back(r); //将新的value入栈
                        Quaternion newq(o, a1, a2, r);
                        Q.push_back(newq);
                    }
                }
                else if(cur_act == 8)//不产生四元式，(E)
                {
                    value.pop_back();
                    a1 = value.back();
                    value.pop_back();
                    value.pop_back();
                    value.push_back(a1);

                }

                //下面为analysis和state的分析过程
                for (int i = 0; i < r_len; i++)  //将产生式长度的字符出容器
                {
                    analysis.pop_back();
                    state.pop_back();
                }
                if (r_vn == 1)  //归约出S来，对应的新状态是acc
                    analysis.push_back(r_vn);  //VN入栈
                else
                {
                    cur_state = state.back(); //获得出栈后的新最新状态
                    cur_state = GoTo[cur_state].row[r_vn - 2];  //加入归约后的VN之后的新状态
                    analysis.push_back(r_vn);  //VN入栈
                    state.push_back(cur_state);  //新状态入栈
```

```
            }

            is_r = true; //此轮发生了归约， 下一轮current就还是当前这一轮的，不再读新字符。


        }
        else if (cur_act == -1)  //Action表中无内容，报错
        {
            cout << "分析出错" << endl;
            return false;
        }
    }
    else  //是VN，查Goto表
    {
        cur_act = Action[cur_state].row[current - 2].action; //要减2，因为Goto表中从S开始，不是S'
        if (cur_act != -1) //Goto表只有移进
        {
            state.push_back(cur_act);
            analysis.push_back(current);
        }
            else  //Goto表中无内容，报错
            {
                cout << "分析出错" << endl;
                return false;
            }
            is_r = false;
        }
    }
    cout << "分析成功" << endl;
    return false;
}
```

程序测试

测试样例1

$$i=a*(b+c)/d\#$$

INPUT:

```
(14,"i")
(7,"=")
(14,"a")
(10,"*")
(12,"(")
(14,"b")
(8,"+")
(14,"c")
(13,")")
(11,"/")
(14,"d")
(15,"#")
```

运行结果如下：

```
分析成功
(+,b,c,A)
(*,a,A,B)
(/,B,d,C)
(=,C,,i)
```