编译器设计专题实验
实验报告

实验 5 语义分析

班级：计算机 2102
姓名：李芝塬
学号：2216113163

目录

实验要求 1
实验分析 1
实验过程 2
代码实现 3
实验结果 4
实验总结 5
附录 6

实验要求

构建语法制导的语义分析程序能在语法分析的同时生成符号表和中间语言代码，并输出结果到文件中。

1. SLR(1)制导的语义分析框架实现
2. 中间语言代码形式，三元式或四元式，或逆波兰表达式

实验分析

本次实验有 2 个要求：完成 SLR(1)制导的语义分析框架，以中间语言代码形式输出。其中 SLR(1)语义分析框架已经在实验 4 中完成，本次实验需要在实验 4 的基础上增加代码，以输出中间语言代码。

考虑到四元式需要生成临时变量， 三元式需要考虑式子的编号，比较复杂，所以使用逆波兰表达式输出。这种表达式在读取时也比较方便，使用栈即可实现。

对于表达式 a+(b+c)*d，输出结果应当为 abc+d*+。

实验过程

1. 对于一个待归纳的式子，从头到尾读取，并按照 SLR(1)分析表进行移进或归约。这一部分已经在实验 4 中实现。
2. 在移进的过程中添加变量名，在归约的过程中添加运算符。
3. 在每次移进一个字符时，先判断这个字符的类型。如果是变量名，则直接添加到待输出的逆波兰表达式的末尾；如果是运算符，则添加到运算符栈的栈顶；如果是括号，则直接忽略。
4. 每次归约时，如果不含运算符，例如 E->T，则直接归约即可；如果有运算符，则需要将对应的符号从栈顶弹出，添加到逆波兰表达式的末尾。
5. 考虑到实际需求，并不需要使用运算符栈存储所有的运算符，而是在归约时直接从产生式中读取即可。

以下是一个例子：

对于 a+(b+c)*d:

1. 读取 a，在逆波兰表达式中添加 a
2. a 被归约为 E,由于归约所用的文法不含运算符，所以不用考虑符号
3. 读取符号+， 忽略。
4. 读取括号，忽略
5. 读取 b，归约为 E，将 b 添加到逆波兰表达式，此时表达式为 ab
6. 读取+， 忽略
7. 读取 c，归约为 T，将 b 添加到逆波兰表达式，此时表达式为 abc
8. b+c 归约为 E，在逆波兰表达式末尾添加+， 表达式为 abc+
9. 读取*和 d，逆波兰表达式变为 abc+d*
10. 最后归约，在结尾添加+， 所以输出的逆波兰表达式为 abc+d*+

所以在实现代码的过程中，移进时遇到变量名则直接添加到逆波兰表达式结尾，遇到括号忽略，遇到符号忽略；归约时，将产生式右部的运算符添加到逆波兰表达式结尾。

代码实现

```
Content &act = action[top][u];
        if (act.type == 0) // shift
        {
            if (u == '*' || u == '+' || u=='(' || u==')'){}
           // 忽略运算符和括号
            else
            {
                // 将变量名添加到逆波兰表达式的结尾
                reverse_Polish_notation += u;
            }
            print(get_steps(steps++), get_stk(op_stack), src.substr(i),
"shift", get_stk(st_stack), act.out, "");
            op_stack.push_back(u);
            st_stack.push_back(act.num);
        }
        else if (act.type == 1) // reduce
        {
            if(act.num==1){ // 第1个文法是E->E+T，对应+
                reverse_Polish_notation+='+';
            }
            else if(act.num==3){ // 第3个文法是T->T*F，对应*
                reverse_Polish_notation+='*';
            }
            WF &tt = wf[act.num];
            int y = st_stack[st_stack.size() - tt.right.length() - 1];
// op-stack弹出右侧后剩余的符号
            int x = Goto[y][tt.left[0]];
            print(get_steps(steps++), get_stk(op_stack), src.substr(i),
get_shift(tt), get_stk(st_stack), act.out, get_steps(x));
            for (int j = 0; j < tt.right.length(); j++)
            {
                st_stack.pop_back();
                op_stack.pop_back();
            }
            op_stack.push_back(tt.left[0]);
            st_stack.push_back(x);
            i--;
        }
```

其中 act 表示 action 表和 goto 表中的规则，act.type 表示是归约还是移进；act.num 表示归约使用的产生式的编号。

在分析完成之后，输出即可。

实验结果

```
现在的日期:
Mon Jul  1 02:25:40 PM CST 2024
开始编译
开始运行
输入的内容:
S
10
S->E
E->E+T
E->T
T->T*F
T->F
F->(E)
F->a
F->b
F->c
F->d
a+(b+c)*d
```

steps	op-stack	input	operation	state-stack	ACTION	GOTO
1	#	a+(b+c)*d#	shift	0	S5	
2	#a	+(b+c)*d#	reduce(F->a)	05	R6	3
3	#F	+(b+c)*d#	reduce(T->F)	03	R4	4
4	#T	+(b+c)*d#	reduce(E->T)	04	R2	2
5	#E	+(b+c)*d#	shift	02	S10	
6	#E+	(b+c)*d#	shift	0210	S1	
7	#E+(	b+c)*d#	shift	02101	S6	
8	#E+(b	+c)*d#	reduce(F->b)	021016	R7	3
9	#E+(F	+c)*d#	reduce(T->F)	021013	R4	4
10	#E+(T	+c)*d#	reduce(E->T)	021014	R2	9
11	#E+(E	+c)*d#	shift	021019	S10	
12	#E+(E+	c)*d#	shift	02101910	S7	
13	#E+(E+c	)*d#	reduce(F->c)	021019107	R8	3
14	#E+(E+F	)*d#	reduce(T->F)	021019103	R4	13
15	#E+(E+T	)*d#	reduce(E->E+T)	0210191013	R1	9
16	#E+(E	)*d#	shift	021019	S12	
17	#E+(E)	*d#	reduce(F->(E))	02101912	R5	3
18	#E+F	*d#	reduce(T->F)	02103	R4	13
19	#E+T	*d#	shift	021013	S11	
20	#E+T*	d#	shift	02101311	S8	
21	#E+T*d	#	reduce(F->d)	021013118	R9	14
22	#E+T*F	#	reduce(T->T*F)	0210131114	R3	13
23	#E+T	#	reduce(E->E+T)	021013	R1	2
24	#E	#	accept	02	acc	

```
逆波兰表达式:
abc+d*+
```

如图，输入 a+(b+c)*d，输出的内容是 abc+d*+

如果输入 a+b+c+d+a,输出为：

```
逆波兰表达式:
ab+c+d+a+
```

符合预期。

实验总结

本次实验需要在上一次实验的基础上进行修改，难度不大，工作量也不大。本次实验较好地完成了实验目标，可以输出逆波兰表达式。

本次实验的不足之处在于，这个实验是针对特定的文法写的，如果换成其他的文法，则会失效。

附录

源代码：Github