



北京交通大学
BEIJING JIAOTONG UNIVERSITY

《编译原理》 实验报告

实验名称： 基于 SLR(1)分析法的语法制导翻译及中间代
码生成程序设计原理与实现

学 号：

姓 名：

学 院：

日 期：

计算机与信息技术学院

2022 年 12 月 07 日

目录

1.	实验目的	3
2.	实验要求	3
3.	程序实现	3
3.1.	相关环境介绍	3
3.2.	主要数据结构	3
3.2.1.	Keyword	3
3.2.2.	Configresult	4
3.2.3.	Expression	4
3.2.4.	State_DFA	6
3.2.5.	Table	8
3.2.6.	其他重要数据结构	8
3.3.	程序结构描述	9
3.3.1.	设计方法	9
3.3.2.	函数定义	9
4.	程序测试	10
5.	实验汇总	15
5.1.	技术难点及解决方案	15
5.2.	实验感想和经验总结	16

1. 实验目的

通过实验，完成赋值语句 SLR (1) 文法语法制导生成中间代码四元式的过程。

$G[S]: S \rightarrow V = E$

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid i$

$V \rightarrow i$

2. 实验要求

- (1) 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；
- (2) 设计中间代码四元式的结构；
- (3) 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；
- (4) 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。(5) 考虑根据文法自动构造 SLR (1) 分析表，并添加到你的程序中。

3. 程序实现

3.1. 相关环境介绍

操作系统：window 10 21H2

开发环境：Clion-2022.2.1-Windows

编译器：mwing-10.0

3.2. 主要数据结构

3.2.1. Keyword

主要是单词的信息保存，建立了一个 struct

```
01: struct Keyword{
02:     string notation;
03:     int class_num;
04:     int line;
```

```

05:   Keyword(string str, int num, int line_){
06:       notation = str;
07:       class_num = num;
08:       line = line_;
09:   }
10:   Keyword(char* str, int num, int line_){
11:       notation = string(str);
12:       class_num = num;
13:       line = line_;
14:   }
15:   Keyword(char str, int num, int line_){
16:       notation = str;
17:       class_num = num;
18:       line = line_;
19:   }
20:};

```

其中 notation 为单词的值，class_num 为单词所属的类别，line 是单词在源程序中的行号。利用 map 容器，制作了以 string 为 index 的二维数组。

3.2.2. Configresult

配合 config_init 函数的返回值，配置了一个对应的结构体 **ConfigResult**。

```

21:struct ConfigResult{
22:    string need_handle;
23:    string output_four;
24:    int mode;
25:};

```

其中，need_handle 为需要处理的输入文件路径；ouput_four 为输出的中间代码四元式文件路径；mode 为 SLR 分析程序的模型，其中 0，只是分析语法正确与否，1 同时输出中间代码四元式。

3.2.3. Expression

针对 SLR 中的 DFA 加点的情况，我们基于产生式的形式，构建了一个 DFA 中使用的产生式的类 **expression**。

```

26:class expression{
27:public:
28:    string left;

```

```
29:     vector<string> right;
30:     int point;
31:
32:     expression(string left_, vector<string> right_, int point_) {
33:         left = left_;
34:         right = right_;
35:         point = point_;
36:     };
37:     bool operator < (const expression &tem) const{
38:         if(left != tem.left) return left < tem.left;
39:         else if(right != tem.right) return right < tem.right;
40:         else return point < tem.point;
41:     }
42:
43:     string get_pointing() const {
44://         return the next element of the right of point
45:         if(point+1 > right.size()) return "end";
46:         else return right[point];
47:     }
48:
49:     expression reason(){
50:         expression tem = *this;
51:         if(point<right.size()){
52:             tem.point++;
53:         }
54:         else tem.point=-1;
55:         return tem;
56:     }
57:
58:     int isend(){
59:         if(point >= right.size()) return 1;
60:         else return 0;
61:     }
62:
63:     bool operator !=(const expression & new_express) const{
64:         if(left != new_express.left || point!= new_express.point ||
right!=new_express.right){
65:             return true;
66:         }
67:         else return false;
68:     }
69:};
```

其中, left 为产生式左部, right 为产生式右部, point 代表点的位置。Expression 类配套了很多函数。重载的小于函数为配合 set 容器排序使用, 后边介绍。Get_pointing 函数得到下一个点指向的符号, 或者得到表达式末尾的提示。Reason 函数在 DFA 创建时推理之用, 返回一个点移动一位的 expression。Isend 函数判断是否推理到末尾, 帮助判断 是否规约。重载的不等于函数也是在构建 DFA 状态时判断之用。

3.2.4. State_DFA

对 DFA 构建中的状态, 构建了一个 **state_DFA** 的类。

```

70: struct state_DFA{
71://    end 1 for have reasoned, so skip it
72:    int id;
73:    set<expression> compont;
74:    map<string, int> next;
75:    string from;
76:    int end=0;
77:
78://from for the input when reasoning
79:    state_DFA(int choice, string left = "NULL"){
80://        choice is for different construction method
81://        0 for Vn closure,
82://        1 for reasoning and simple
83:        if(choice ==0){
84:            for(auto&item:grammar[left]){
85:                if(item.empty()) break;
86:                expression tem(left, item, 0);
87:                compont.insert(tem);
88:            }
89:            int flag=1;
90:            while(flag){
91:                int length = compont.size();
92:                for(auto item:compont){
93:                    string tem_point = item.get_pointing();
94:                    if(judge_Vn(tem_point)){
95:                        for(auto item1:grammar[tem_point]){
96:                            if(item1.empty()) break;
97:                            expression tem_express(tem_point, item1,
0);
98:                            compont.insert(tem_express);
99:                        }
100:                    }
101:                }
102:                if(length == compont.size()) flag=0;

```

```
103:         }
104:     }
105:     else if(choice==1){
106:         from = left;
107:     }
108:     else if(choice == 2){
109://         do nothing
110:
111:     }
112: }
113:
114:
115:
116: bool operator < (const state_DFA &tem) const{
117:     if(compont.size() != tem.compont.size()){
118:         return compont.size()<tem.compont.size();
119:     }
120:     else{
121:         for(auto tem1 = compont.begin(), tem2 =
tem.compont.begin(); tem1!=compont.end(); ++tem1, ++tem2){
122:             return *tem1 < *tem2;
123:         }
124:     }
125: }
126:
127: int get_next(){
128:     for(auto &item:compont){
129:         string point_now = item.get_pointing();
130:         if(point_now=="end") continue;
131:         else{
132:             expression tem_express(item);
133:             tem_express.point++;
134:         }
135:
136:     }
137: }
138:
139: bool operator == (const state_DFA & new_state) const{
140:     if(compont.size() != new_state.compont.size()) return false;
141:     else {
142:         for(auto i=compont.begin(), j = new_state.compont.begin();
i!=compont.end(); i++, j++){
143:             if(*i != *j){
```

```

144:                return false;
145:            }
146:        }
147:        return true;
148:    }
149: }
150:
151:
152:};

```

Id 为状态的标号。Compont 为由 expression 组成的集合，在构建 DFA 的函数中需要判断表达式是否重合的功能，直接使用集合用来自动判断，类的集合只需要重载一个小于函数用以排序即可，这在 expression 类中实现了。from 标记了该状态的上一个状态读入了什么字符来到了该状态，用来判重。end 为状态推理完毕标记。

3.2.5. Table

分析表的单位 table

```

153:struct table{
154://    0 for move inside, 1 for jump goto, 2 for acc, 3 for guiyue
155:    int next;
156:    int op;
157:    string left;
158:    vector<string> right;
159:};

```

Next 为读入的下一个字符对应的状态号；op 对应四个操作，具体见代码注释；对于需要规约的操作，有表达式的简单结构左部和右部。

3.2.6. 其他重要数据结构

下边是以 stl 为基础建立的相关数据结构

```

160:map<string, map<string, table>> analyzer_table;
161:map<string, string> Vn;
162:map<string, string> Vt;
163:map<string, set<string> > first;
164:map<string, set<string> > follow;
165:vector<state_DFA> DFA_set;
166:map<string, vector<string>[ORMAXFORGRAMMAR] > grammar;
167:

```

analyzer_table 是分析表，包括所有的操作和相应的推理表达式等等。同时，考虑到

SLR 分析法相对于算符优先文法的分析表虽然稠密了不少，但是仍为稀疏矩阵，我们直接采用了行列分开的方法，每行只保留了有数值的列号，这样可以减少空间占用。

Vn、Vt、first、follow 集的作用见前几个实验。

DFA_set 即有限自动机所有状态集合。

Grammar 为文法的存储结构，文法可以写入 grammar.txt 文件中，由相关函数读入，而不是固化在代码中，增加了鲁棒性。

使用 map 结构获得由字符串结构作为类似 index 的表结构替代是实验 3 中的成果。这里使用 map 结构完成了 ppt 中讲授的以 Vt 或者 Vn 符号作为二维数组 index。

3.3. 程序结构描述

3.3.1. 设计方法

考虑到程序的通用性，程序的输入流全部都是 txt 文件。首先读入 grammar 文件，其中包括文法和 Vn 以及 Vt 集的定义。之后，在程序中创建 Vn 集合 Vt 集，然后求的 first 和 follow 集。在这基础上构建 DFA 和分析表程序。构建成功后，按格式化输出，这样可以方便手动验证正确性。

接下来是和词法分析程序的联动，本程序可以直接调用词法分析程序，因此测试样例直接准备源代码待分析即可。这其实是实验 6 的要求，但其实在实验 2 开始，程序的设计理念就是完整前端，并且所有成熟模型一直沿用。

对于 SLR 分析法，本人自信已经完成的十分健壮，可以直接修改 grammar 中的文件更改文法，完成不同 SLR 文法的语法判断；但同时，对于中间文法的生成，本程序只实现了赋值文法的生成，为了不影响其他非赋值文法的语法分析，我们加入了 mode 选项，可以关闭中间代码的生成，符合开闭原则。

本文大量引入面向对象编程思想、相应设计模式思想和单元测试思想，在前几次实验编码的基础上，大量进行函数封装，以类为中心，尽量使用类成员函数完成编写。

当然，语法分析部分的输入当然还是词法分析的结果，也就是二元式，同时本程序为了报错的方便，在二元式的基础上加入了行号 line，变成了三元式。其中的相应 config 配置依赖文件参考实验 1，包括各种文件依赖。

对于中间代码的输出，首先是逆波兰式，后边才是每一个四元式计算，临时变量采用 T 加上下标的形式。

程序的依赖文件和待分析的字符创文件，在 config 文件中进行更改。对于词法分析文件的依赖见实验 1 技术文档。

3.3.2. 函数定义

create_Vn_Vt_grammar(grammar_path); 创建 Vn Vt 集和文法表。

create_first(); 创建 first 集。

create_last(); 创建 lastVt 集。

create_analysis_table(); 创建分析表

analyzer_stack(need_to_analysis); 分析栈入口，即分析驱动程序。

int correct_prom() 语句符合文法提示。

int error_prom() 语句不合文法提示。

DFA_state_show() DFA 输出打印函数。

Analysis_table_show() 分析表输出打印函数。

还有一众辅助函数，不表。

主要函数的编写参照课程讲述。

基本函数和课堂 ppt 实现大致相同，流程图略去。

规约的产生式右部寻找与报错相对为原创。原理为暴力遍历。

4. 程序测试

Alu_test1.txt

这里一个文件中包含了 4 个测试样例。

168:

169:k=a- b * c #

170:223 = 23434+34324+(a*b / c) +(a*b/ (2-2)) #

171:a = (a+ b -c) * 32 / a #

172:b=(a+c * b* #

这是本程序的最初输入，由于是从词法分析开始，下边给出中间的结果的输出：

notation	class_num	line	
k	0	1	
=	26	1	
a	0	1	
-	36	1	
b	0	1	
*	39	1	
c	0	1	
#	24	1	
223	1	2	I
=	26	2	
23434	1	2	
+	33	2	
34324	1	2	
+	33	2	
(45	2	
a	0	2	
*	39	2	
b	0	2	
/	42	2	

图表 4-1 test1 中词法分析结果

alu_class_base_result.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

notation	class_num	line
+	33	1
-	36	1
*	39	1
/	42	1
(45	1
)	46	1
#	24	1
=	26	1

图表 4-2 符号类号文件（词法分析程序的输出结果的一部分）

```

base_source ../alu_class_base_test.txt
base_out ../alu_class_base_result.txt
input_file ../alu_test1.txt
output_file ../alu_result1.txt
output_four ../four_elements.txt
four_output_mode 1

end

// PATH have something problems
// path have blank so fin stop after junior
// path recommend to use compared path case the absolute path has blank
// base on reversed_word.txt
// output_four is path of output file of four element formula
// mode 0 not output four element formula, mode 1 instead
// config reserved_word grammar alu_class_base_test files is necessary

```

图表 4-3 config 文件

其中包括了一众依赖文件的路径还有默认待分析文件路径，最下边是书写格式。相比于前几个实验，config 文件多了四元式输出文件和模式选项。

```

Vn E T F V S
Vt + - * / = ( ) i n #

S -> V = E
E -> E + T | E - T | T
T -> T * F | T / F | F
F -> ( E ) | i | n
V -> i

start S
end
// e is epilog
// each line ends with nothing, no blank
// Vt i is identifier num

```

图表 4-4 grammar 文件

Grammar 文件主要是 Vn Vt 集还有文法的输入，下方是特殊说明。同时，由于在分析表中有接受操作，所以要特殊指定开始符号。同时，我们对文法进行了拓展，等号右部可以出现数字 n，原来为只有标识符 i。

词法分析程序输出仍然采用了三元组，多了一个行号，方便错误提示，更多的我们直接用了 Keyword 结构，这对结果没有影响。

```
line 1 syntax correct
line 2 has syntax error
line 3 syntax correct
line 4 has syntax error

Process finished with exit code 0
```

图表 4-5 本程序结果，信息提示

通过信息提示可以看出，程序对 4 个测试样例完成的很好，结果正确。等式中，实验指导书原文法标识符位置，可以出现数字常量。这是对文法的拓展。同时为了方便判断程序正确与否，还输出了 DFA 和分析表。

```
id      0
S -> . V = E
V -> . i

next

V      1
i      2

id      1
S -> V . = E

next

=      3

id      2
V -> i .

next

id      3
E -> . E + T
E -> . E - T
```

图表 4-6 DFA 结果

DFA 输出的结果包括状态号，产生式情况，还有不同输入对应的下一个状态。

analysis_table is following
0 for move inside, 1 for jump goto, 2 for acc, 3 for guiyue

state_id	input	op
0		
	V	1
	i	0
1		
	=	0
2		
	=	3
3		
	(0
	E	1
	F	1
	T	1
	i	0
	n	0
4		
	#	2
	+	0
	-	0
5		
	#	3
)	3
	*	0
	+	3
	-	3
	/	0

图表 4-7 SLR 分析表

分析表包括状态号，不同输入的操作，分 4 种不同操作，对应操作号。最后一列为相应的补充，如 goto 操作则是转向的状态；规约为使用的产生式。

alu_test1.txt × four_elements.txt × alu_result1.txt × grammar.txt ×

line 1
k a b c * - =

op	left	right	result
*	b	c	T0
-	a	T0	T1
=	k	T1	T2

line 3
a a b + c - 32 * a / =

op	left	right	result
+	a	b	T0
-	T0	c	T1
*	T1	32	T2
/	T2	a	T3
=	a	T3	T4

图表 4-8 四元式的输出

输出包括行号、逆波兰式、四元式。由于是追加写模式打开文件，所以上次的分析结果不会被覆盖。

5. 实验汇总

5.1. 技术难点及解决方案

实验本身的重要程序框图，已经在课堂上教授过了，主要是代码编写过程中的问题有些多，所幸最后都解决了。

实验代码编写完成之后，本人突然发现，实验指导的文法与正常编程语言的语法逻辑还是有一定区别的。其中 *i* 被定义为标识符，也就是变量，这在算法表达式中 *i* 出现的位置可以是标识符也可以是数字常量，甚至可以是一个表达式。这里我们在原有文法的基础上做了一点拓展，新增了一个 *n* 的 *Vt* 符号，作为数字常量的代指。这样我们的文法就变成了：幸运的是，在修改文法后本人的程序居然没有一点问题，这真是值得欣慰的事。

$$\begin{aligned} S &\rightarrow V = E \\ E &\rightarrow E + T \mid E - T \mid T \\ T &\rightarrow T * F \mid T / F \mid F \\ F &\rightarrow (E) \mid i \mid n \\ V &\rightarrow i \end{aligned}$$

当然其实我们也可以把 i 再拓展为表达式，这样其实是一种递归调用。

实验最难的地方，还是相应数据结构的创建，本程序主要使用的基于 STL 的数据机构。

5.2. 实验感想和经验总结

这次实验主要的编码困难在于数据结构的建立。C++ 中的二维数组 index 都是整数，课堂上教授的分析表等表格都是字符作为 index，这其中的转换是有难度的。最后，想到了使用 STL 中的 map 容器作为数据结构，二维数组就是 map 套 map，还用了 vector 来存储文法产生式，这样就会出现“数据结构”一节中看似很复杂的容器嵌套结构。数据结构敲定之后，再定义出基本的操作方法，算法沿用课程中教授的，实验就不难完成了。

Clion debug 变量监视，如果在变量创建之前开始监视，会出现问题。同时，在大量中间结构化结果需要观测时，如 DFA 和分析表的创建与否，我们可以采用格式化打印函数来完成。对于直接写输出函数进行中间结果的查看还是进行借助 IDE debug 进行可视化查看有了更深的理解。在许多 IDE 可视化效果不太好，或者说需要点击的次数太多时，建议输出函数，直接层层格式化输出，这样可以自定义格式还可以提高自己的 debug 水平。

一开始用 map 作为容器，可以试二维数组 index 为字符。现在想来也可以自定义结构体，或者直接面向对象写一个类和相关函数出来，用轮子保证可用性。

状态 state——DFA 使用了 set，需要重载 operator <。

面向对象编程的大量使用，使得许多函数的创建有迹可循，更加方便 debug。许多地方可以再使用一下面向对象编程的思想，多把过程封装为类函数。重载的 != 函数 逻辑实则为 ==，逻辑容易颠倒。

同时，分析表的数据结构中含有产生式的右部，其实可以简化为一个左部和右部的个数。

使用 for(auto &item: DFA_set) 进行遍历，同时发生了 vector 内部的修改，触发底层移动机制，整个 vector 移动，原本的 item 指针访问内存失败。通过这个例子，本人对 item 方法遍历和用下标进行遍历有了更深的理解。下标遍历由 vector 标号进入，这种方法是不会受到原本内存移动影响，或者说，标号的地址是操作系统自动维护的；通过提前定义引用变量进行访问，当原本指针所指的内存变化时，是不会自动维护的，需要由相关代码完成。同时，我们可以知道 for auto 方法遍历确实更加快捷，但是也会遇到各种更加底层的问题，主要还是在只读模式下进行。

一开始，对于 DFA 的推理，也想过由递归的方式进行，也就是由一个状态里面的一个式子，从头推到尾；然后进行回溯，发现其他式子继续推理。后来发现实现起来不太好，同时对于状态的合并不好处理，于是采取了直接动态数组加入的方式进行。

运行采用了软件工程的单元测试思想，分阶段测试函数结果，于是在每个重要函数后边加入了结果可视化函数，这样比 debug 更快的指导结果。

考虑到 LR (1) 分析表虽然相比自底向上分析表要稠密不少，但是仍然是一个稀疏矩阵，于是分行，按列进行组织，这一列有数据方保存，error 不保存，相当于，找不到是 error。

实验很多轮子是基于前几个实验的，更加明白了开闭原则。原来的代码不能随便改。有时候会误更改，这也是本实验遗憾的地方，没有把成熟的可以沿用的模型编程头文件进行引用，而是直接拷贝进新实验的代码中。