

语法分析程序的设计与实现

2017211305 班 2017211240 于海鑫

版本: ϵ

更新: November 27, 2019

本文档为编译原理课程实验“语法分析程序的设计与实现”的实验报告。

1 实验题目

1.1 题目

语法程序的设计与实现

1.2 实验内容

编写语法分析程序，实现对算数表达式的语法解析。我们要分析的文法如下：

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

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

$$F \rightarrow (E) \mid id$$

1.3 实验要求

在对输入表达式进行分析的过程中，输出所采用的生成式。使用如下方法：

- 编写 $LL(1)$ 语法分析程序，要求如下：
 - (1). 编程实现算法 4.2，为给定文法自动构造预测分析表
 - (2). 编程实现算法 4.1，构造 $LL(1)$ 预测分析程序

- 编写语法分析程序实现自底向上的分析，要求如下：
 - (1). 构造识别所有或前缀的 DFA
 - (2). 构造 LR 分析表
 - (3). 编程实现算法 4.3，构造 LR 分析程序

2 实验分析

本次实验我们实现了如上所述两种方法。其中使用的词法分析使用了词法分析类，我们使用的词法分析类如下：

```
1 class Lexer:
2     def __init__(self, expression):
3         self.token = None
4         self.expression = (expression + '$').replace("_", "")
5
6     def next_token(self):
7         if self.token:
8             temp = self.token
9             self.token = None
10            return temp
11        else:
12            return self._next_token()
13
14    def peek_token(self):
15        if not self.token:
16            self.token = self._next_token()
17        return self.token
18
19    def _next_token(self):
20        if self.expression == '':
21            return None
22        c = self.expression[0]
23        if c == '(' or c == ')' or c == '+' or c == '-' or c == '*'
24            or c == '/' or c == '$':
25            self.expression = self.expression[1:]
26            return c
27        else:
28            result = None
```

```

28         i = 1
29         try:
30             while i < len(self.expression):
31                 if self.expression[i - 1] == '.':
32                     i += 1
33                     result = float(self.expression[:i])
34                     i += 1
35             except:
36                 self.expression = self.expression[i - 1:]
37                 return 'n'#result
38         t = len(self.expression)
39         self.expression = self.expression[i - 1:]
40         return 'n' if i == t else result

```

3 $LL(1)$ 语法分析

3.1 实验原理

构造分析表的方法如图 1 所示。

INPUT: Grammar G .

OUTPUT: Parsing table M .

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $FIRST(\alpha)$, then for each terminal b in $FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $FIRST(\alpha)$ and $\$$ is in $FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

图 1: 构造分析表

在构造分析表之前，我们还需要进行几个步骤：

- (1). 消除左递归
- (2). 构造 $FIRST$ 集, $FOLLOW$ 集

所谓消除左递归，就是类似将 $E \rightarrow E + T \mid T$ 转化为 $E \rightarrow TE'$ 以及 $E' \rightarrow +TE' \mid \epsilon$ 的操作，这部分很容易使用编程语言实现。

计算 $FIRST$ 集以及 $FOLLOW$ 集则按照定义计算即可。

在获取了预测表之后，我们就可以使用如算法 4.1 所述的方式进行语法解析。

3.2 数据结构

为了实现预测分析程序，我们需要设计一套数据结构用以保存语法、预测分析表以及转换的状态。

3.2.1 语法

对于语法，我们需要保存其生成式以及起始符号等信息，其定义如下：

```
1 class Grammar:
2     def __init__(self):
3         self.startSymbol = 'S' # default start symbol
4         self.nonTerminalSymbol = set()
5         self.terminalSymbol = set()
6         self.generatorExpression = {}
7
8         # Temp variable
9         self.nullable = {}
10        self.firstSymbols = {}
11        self.followSymbols = {}
12        self.llTable = {}
```

3.2.2 预测分析表

预测分析表实际上是一个二维字典，不再详细介绍。

3.2.3 转换程序

在转换程序内，我们要保存之前生成的预测分析表以及分析时所需的栈，定义如下：

```
1 class LLParser:
2     def __init__(self, grammar, expression):
3         self.grammar = grammar
4         self.lexer = Lexer(expression)
5         self.stack = ['$ ', self.grammar.startSymbol]
```

其中的 `lexer` 是我们的词法分析器。

3.3 算法实现

实现与书上一致。主要代码如下：

```
1 class Grammer:
2     def __init__(self):
3         self.startSymbol = 'S' # default start symbol
4         self.nonTerminalSymbol = set()
5         self.terminalSymbol = set()
6         self.generatorExpression = {}
7
8         # Temp variable
9         self.nullable = {}
10        self.firstSymbols = {}
11        self.followSymbols = {}
12        self.llTable = {}
13
14    def get_unused_non_terminal_symbol(self):
15        tmp = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
16        for i in tmp:
17            if i not in self.nonTerminalSymbol:
18                self.nonTerminalSymbol.add(i)
19            return i
20
21    def solve_left_recursion(self):
22        keys = list(self.generatorExpression.keys())
23        for k in keys:
24            v = self.generatorExpression[k]
25            leftRecu = [i for i in v if i[0] == k]
26            if leftRecu:
27                newSymbol = self.get_unused_non_terminal_symbol()
28                self.generatorExpression[k] = [
29                    i + newSymbol for i in v if i[0] != k]
30                self.generatorExpression[newSymbol] = [
31                    i[1:] + newSymbol for i in leftRecu]
32                self.generatorExpression[newSymbol].append('')
33
34    def compute_first_and_follow_set(self):
35        old_nullable = {}
36        for k in self.nonTerminalSymbol:
```

```

37         self.nullable[k] = False
38     while old_nullable != self.nullable:
39         old_nullable = self.nullable.copy()
40         for k, v in self.generatorExpression.items():
41             self.nullable[k] = False
42             for expressions in v:
43                 nullable = True
44                 for i in expressions:
45                     if not (i in self.nonTerminalSymbol and '' in
46                             self.generatorExpression[i]):
47                         nullable = False
48                 self.nullable[k] = self.nullable[k] or nullable
49
50     old_first_set = set()
51     for k in self.nonTerminalSymbol:
52         self.firstSymbols[k] = set()
53     for k, v in self.generatorExpression.items():
54         for expressions in v:
55             for i in expressions:
56                 if i not in self.nonTerminalSymbol:
57                     self.firstSymbols[i] = set(i)
58     while (old_first_set != self.firstSymbols):
59         old_first_set = self.firstSymbols.copy()
60         for k, v in self.generatorExpression.items():
61             for expressions in v:
62                 add_new_first = True
63                 if expressions == '':
64                     self.firstSymbols[k].add('')
65                 for i in expressions:
66                     if add_new_first:
67                         self.firstSymbols[k] = self.firstSymbols[
68                             k].union(
69                                 self.firstSymbols[i])
70                     add_new_first = add_new_first and (
71                         i in self.nonTerminalSymbol and self.
72                             nullable[i])
73
74     old_follow_set = set()
75     for k in self.nonTerminalSymbol:

```

```

73         self.followSymbols[k] = set()
74     for k, v in self.generatorExpression.items():
75         for expressions in v:
76             for i in expressions:
77                 if i not in self.nonTerminalSymbol:
78                     self.followSymbols[i] = set(i)
79     self.followSymbols[self.startSymbol].add('$')
80     while(old_follow_set != self.followSymbols):
81         old_follow_set = self.followSymbols.copy()
82         # print(self.generatorExpression.items())
83         for key, v in self.generatorExpression.items():
84             for expressions in v:
85                 for i in range(len(expressions)):
86                     add_follow = True
87                     for j in range(i + 1, len(expressions)):
88                         add_follow = add_follow and (
89                             expressions[j] in self.
90                                 nonTerminalSymbol and self.
91                                 nullable[expressions[j]])
92                     if add_follow:
93                         self.followSymbols[
94                             expressions[i]] = self.followSymbols[
95                                 expressions[i]].union(
96                                     self.followSymbols[key])
97                     for j in range(i + 1, len(expressions)):
98                         add_follow = True
99                         for k in range(i + 1, j):
100                             add_follow = add_follow and (
101                                 expressions[k] in self.
102                                     nonTerminalSymbol and self.
103                                     nullable[expressions[k]])
104                     if add_follow:
105                         self.followSymbols[expressions[i]] =
106                             self.followSymbols[expressions[i]
107                                 ].union(
108                                     self.firstSymbols[expressions[j]
109                                         ])
110     keys = list(self.firstSymbols.keys())
111     for k in keys:

```

```

105         if k not in self.nonTerminalSymbol:
106             self.firstSymbols.pop(k)
107     keys = list(self.followSymbols.keys())
108     for k in keys:
109         if k not in self.nonTerminalSymbol:
110             self.followSymbols.pop(k)
111     for k, v in self.followSymbols.items():
112         if ' ' in v:
113             v.remove(' ')
114
115     def _get_first(self, expression):
116         if expression == '':
117             return set('')
118         result = set()
119         for i in expression:
120             if i in self.nonTerminalSymbol:
121                 result = result.union(self.firstSymbols[i])
122                 if not self.nullable[i]:
123                     return result
124             else:
125                 return result.union(i)
126         return result
127
128     def get_first(self, expression):
129         result = self._get_first(expression)
130         if ' ' in result:
131             result.remove(' ')
132         return result
133
134     def get_nullable(self, expression):
135         if expression == '':
136             return True
137         return ' ' in self._get_first(expression)
138
139     def build_ll_table(self):
140         for k in self.nonTerminalSymbol:
141             self.llTable[k] = {}
142         for k, v in self.generatorExpression.items():
143             for expressions in v:

```



```

144         for i in self.get_first(expressions):
145             self.llTable[k][i] = expressions
146         if self.get_nullable(expressions):
147             for i in self.followSymbols[k]:
148                 self.llTable[k][i] = expressions
149
150
151 class Lexer:
152     def __init__(self, expression):
153         self.token = None
154         self.expression = (expression + '$').replace("_", "")
155
156     def next_token(self):
157         if self.token:
158             temp = self.token
159             self.token = None
160             return temp
161         else:
162             return self._next_token()
163
164     def peek_token(self):
165         if not self.token:
166             self.token = self._next_token()
167         return self.token
168
169     def _next_token(self):
170         if self.expression == '':
171             return None
172         c = self.expression[0]
173         if c == '(' or c == ')' or c == '+' or c == '-' or c == '*'
174             or c == '/' or c == '$':
175             self.expression = self.expression[1:]
176             return c
177         else:
178             result = None
179             i = 1
180             try:
181                 while i < len(self.expression):
182                     if self.expression[i - 1] == '.':

```

```

182         i += 1
183         result = float(self.expression[:i])
184         i += 1
185     except:
186         self.expression = self.expression[i - 1:]
187         return 'n' # result
188     t = len(self.expression)
189     self.expression = self.expression[i - 1:]
190     return 'n' if i == t else result
191
192
193 class LLParser:
194     def __init__(self, grammer, expression):
195         self.grammer = grammer
196         self.lexer = Lexer(expression)
197         self.stack = ['$ ', self.grammer.startSymbol]
198
199     def _parse(self):
200         if self.stack[-1] not in self.grammer.nonTerminalSymbol:
201             if self.stack[-1] == self.lexer.peek_token():
202                 self.stack.pop()
203                 self.lexer.next_token()
204             else:
205                 raise RuntimeError
206         else:
207             expression = self.grammer.llTable[self.stack[-1]
208                                                         ][self.lexer.peek_token
209                                                         ()]
210
211             print(self.stack[-1] + '→' +
212                   ('\'\'\'\' if expression == \'\' else expression))
213             self.stack.pop()
214             for x in expression[::-1]:
215                 self.stack.append(x)
216
217     def parse(self):
218         while self.stack[-1] != '$':
219             self._parse()

```

```

220 if __name__ == '__main__':
221     expression = "(1+_2)_+_2.5"
222     grammer = Grammer()
223     grammer.startSymbol = 'E'
224     grammer.nonTerminalSymbol.add('E')
225     grammer.generatorExpression['E'] = ['E+T', 'E-T', 'T']
226     grammer.nonTerminalSymbol.add('T')
227     grammer.generatorExpression['T'] = ['T*F', 'T/F', 'F']
228     grammer.nonTerminalSymbol.add('F')
229     grammer.terminalSymbol.add('n')
230     grammer.generatorExpression['F'] = ['(E)', 'n']
231     print(grammer.generatorExpression)
232     grammer.solve_left_recursion()
233     print(grammer.generatorExpression)
234     grammer.compute_first_and_follow_set()
235     print(grammer.nullable)
236     print(grammer.firstSymbols)
237     print(grammer.followSymbols)
238     grammer.build_ll_table()
239     print(grammer.llTable)
240     ll = LLParser(grammer, expression)
241     ll.parse()

```

3.4 运行结果

我们以解析 $(1 + 2) + 2.5$ 为例，程序的输出如下：

```

1 PS D:\playground\FCalculator> python .\LL1\Grammer.py
2 {'E': ['E+T', 'E-T', 'T'], 'T': ['T*F', 'T/F', 'F'], 'F': ['(E)', 'n
  ']}
3 {'E': ['TA'], 'T': ['FB'], 'F': ['(E)', 'n'], 'A': ['+TA', '-TA',
  ''], 'B': ['*FB', '/FB', '']}
4 {'E': False, 'F': False, 'B': True, 'T': False, 'A': True}
5 {'E': {'n', '('}, 'F': {'n', '('}, 'B': {'', '/', '*'}, 'T': {'n',
  '('}, 'A': {'', '+', '-'}}
6 {'E': {')', '$'}, 'F': {'/', '*', ')', '-', '+', '$'}, 'B': {')',
  '-', '+', '$'}, 'T': {')', '-', '+', '$'}, 'A': {')', '$'}}
7 {'E': {'n': 'TA', '(': 'TA'}, 'F': {'(': '(E)', 'n': 'n'}, 'B': {'*':
  '*FB', '/': '/FB', ')': '', '-': '', '+': '', '$': ''}, 'T': {'n

```

```

      ': 'FB', '(' : 'FB'}, 'A': {'+' : '+TA', '-' : '-TA', ')' : '', '$' :
      ''}}
8  E -> TA
9  T -> FB
10 F -> (E)
11 E -> TA
12 T -> FB
13 F -> n
14 B -> ''
15 A -> +TA
16 T -> FB
17 F -> n
18 B -> ''
19 A -> ''
20 B -> ''
21 A -> +TA
22 T -> FB
23 F -> n
24 B -> ''
25 A -> ''

```

其中”表示 ϵ ，输出与预测一致。

4 LR 语法分析

4.1 实验原理

我们首先构造识别文法的 DFA，如图 2。

可以发现，DFA 中存在移进-规约冲突，因此我们使用 *SLR*(1) 进行语法分析，构造的分析表如下一页表格所示。

—

	+	-	x	/	()	id	E	T	F	\$
0					S4		S5	1	2	3	
1	S6	S7									ACC
2	R3	R3	S8	S9		R3					R3
3	R6	R6	R6	R6		R6					R6
4					S4		S5	10	2	3	
5	R8	_____									-
6					S4		S5		11	3	
7					S4		S5		12	3	
8					S4		S5			13	
9					S4		S5			14	
10	S6	S7				S15					
11	R1	R1	S8	S9		R1					R1
12	R2	R2	S8	S9		R2					R2
13	R4	R4	R4	R4		R4					R4
14	R4	R4	R5	R5		R5					R5
15	R7	_____									-

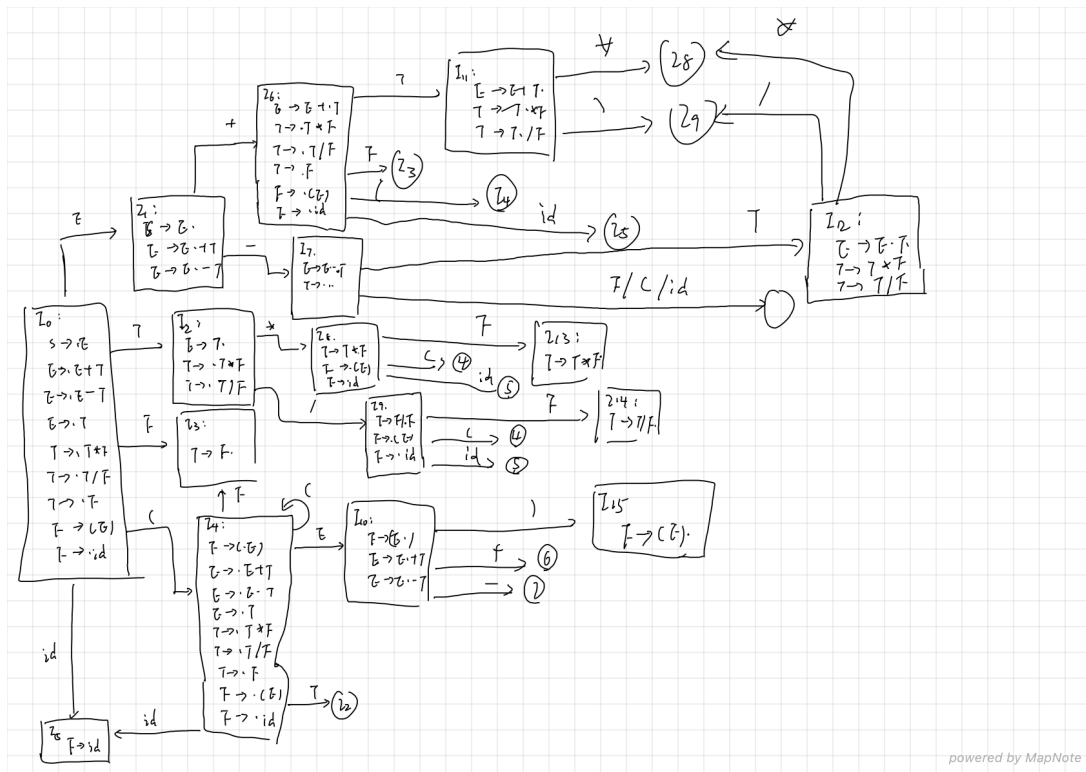


图 2: 构造 DFA

4.2 实验代码

解析算法与书上一致。主要代码如下：

```

1 class Lexer:
2     def __init__(self, expression):
3         self.token = None
4         self.expression = (expression + '$').replace("_", "")
5
6     def next_token(self):
7         if self.token:
8             temp = self.token
9             self.token = None
10            return temp
11        else:
12            return self._next_token()
13
14    def peek_token(self):
15        if not self.token:
16            self.token = self._next_token()
17        return self.token

```

```

18
19 def _next_token(self):
20     if self.expression == '':
21         return None
22     c = self.expression[0]
23     if c == '(' or c == ')' or c == '+' or c == '-' or c == '*'
24         or c == '/' or c == '$':
25         self.expression = self.expression[1:]
26         return c
27     else:
28         result = None
29         i = 1
30         try:
31             while i < len(self.expression):
32                 if self.expression[i - 1] == '.':
33                     i += 1
34                     result = float(self.expression[:i])
35                     i += 1
36             except:
37                 self.expression = self.expression[i - 1:]
38                 return 'n' # result
39     t = len(self.expression)
40     self.expression = self.expression[i - 1:]
41     return 'n' if i == t else result
42
43 class LR1Table:
44     def __init__(self):
45         self.grammer = {}
46         self.action = {}
47         self.goto = {}
48
49
50 class LR1Parser:
51     def __init__(self, table, expression):
52         self.stack = []
53         self.table = table
54         self.lexer = Lexer(expression)
55         self.stack.append((0, ''))

```

```

56
57 def _parse(self):
58     print(self.stack)
59     s = self.stack[-1][0]
60     a = self.lexer.peek_token()
61     print(str(s) + ',' + self.stack[-1][1] + '_--_' + a)
62     action = self.table.action[s][a]
63     if action[0] == 'S':
64         self.stack.append((action[1], a))
65         self.lexer.next_token()
66         return True
67     elif action[0] == 'R':
68         r_expression = self.table.grammar[action[1]]
69         for i in range(len(r_expression[1])):
70             self.stack.pop()
71             if action[1] != 0:
72                 self.stack.append((
73                     self.table.goto[self.stack[-1][0]][r_expression
74                         [0]], r_expression[0]))
75             print(r_expression[0] + '->' +
76                 r_expression[1])
77             return action[1] != 0
78     else:
79         print('Error in parsing')
80         return False
81
82 def parse(self):
83     result = True
84     while (result):
85         result = self._parse()
86
87 if __name__ == '__main__':
88     table = LR1Table()
89     table.grammar = {
90         0: ("S", "E"),
91         1: ("E", "E+T"),
92         2: ("E", "E-T"),
93         3: ("E", "T"),

```



```

94         4: ( "T", "T*F" ),
95         5: ( "T", "T/F" ),
96         6: ( "T", "F" ),
97         7: ( "F", "(E)" ),
98         8: ( "F", "n" )
99     }
100     table.goto = {
101         0: {
102             'E': 1,
103             'T': 2,
104             'F': 3
105         },
106         4: {
107             'E': 10,
108             'T': 2,
109             'F': 3
110         },
111         6: {
112             'T': 11,
113             'F': 3
114         },
115         7: {
116             'T': 12,
117             'F': 3
118         },
119         8: {
120             'F': 13
121         },
122         9: {
123             'F': 14
124         }
125     }
126     table.action = {
127         0: {
128             '(': ('S', 4),
129             'n': ('S', 5)
130         },
131         1: {
132             '+': ('S', 6),

```

```

133         '-': ('S', 7),
134         '$': ('R', 0)
135     },
136     2: {
137         '+': ('R', 3),
138         '-': ('R', 3),
139         '*': ('S', 8),
140         '/': ('S', 9),
141         ')': ('R', 3),
142         '$': ('R', 3)
143     },
144     3: {
145         '+': ('R', 6),
146         '-': ('R', 6),
147         '*': ('R', 6),
148         '/': ('R', 6),
149         ')': ('R', 6),
150         '$': ('R', 6)
151     },
152     4: {
153         '(': ('S', 4),
154         'n': ('S', 5)
155     },
156     5: {
157         '+': ('R', 8),
158         '-': ('R', 8),
159         '*': ('R', 8),
160         '/': ('R', 8),
161         ')': ('R', 8),
162         '$': ('R', 8)
163     },
164     6: {
165         '(': ('S', 4),
166         'n': ('S', 5)
167     },
168     7: {
169         '(': ('S', 4),
170         'n': ('S', 5)
171     },

```

```

172     8: {
173         '(': ('S', 4),
174         'n': ('S', 5)
175     },
176     9: {
177         '(': ('S', 4),
178         'n': ('S', 5)
179     },
180     10: {
181         '+': ('S', 6),
182         '-': ('S', 7),
183         ')': ('S', 15)
184     },
185     11: {
186         '+': ('R', 1),
187         '-': ('R', 1),
188         '*': ('S', 8),
189         '/': ('S', 9),
190         ')': ('R', 1),
191         '$': ('R', 1)
192     },
193     12: {
194         '+': ('R', 2),
195         '-': ('R', 2),
196         '*': ('S', 8),
197         '/': ('S', 9),
198         ')': ('R', 2),
199         '$': ('R', 2)
200     },
201     13: {
202         '+': ('R', 4),
203         '-': ('R', 4),
204         '*': ('R', 4),
205         '/': ('R', 4),
206         ')': ('R', 4),
207         '$': ('R', 4)
208     },
209     14: {
210         '+': ('R', 5),

```

```

211         '-': ('R', 5),
212         '*': ('R', 5),
213         '/': ('R', 5),
214         ')': ('R', 5),
215         '$': ('R', 5)
216     },
217     15: {
218         '+': ('R', 7),
219         '-': ('R', 7),
220         '*': ('R', 7),
221         '/': ('R', 7),
222         ')': ('R', 7),
223         '$': ('R', 7)
224     }
225 }
226 parser = LR1Parser(table, "(1+2)*3-4")
227 parser.parse()

```

4.3 运行结果

我们以解析 $(1 + 2) * 3 - 4$ 为例，程序的输出如下：

```

1 PS D:\playground\FuckingCalculator> python .\LR1\LR1.py
2 [(0, '')]
3 0, -- (
4 [(0, ''), (4, '(')]
5 4,( -- n
6 [(0, ''), (4, '('), (5, 'n')]
7 5,n -- +
8 F->n
9 [(0, ''), (4, '('), (3, 'F')]
10 3,F -- +
11 T->F
12 [(0, ''), (4, '('), (2, 'T')]
13 2,T -- +
14 E->T
15 [(0, ''), (4, '('), (10, 'E')]
16 10,E -- +
17 [(0, ''), (4, '('), (10, 'E'), (6, '+')]

```

```

18 6,+ -- n
19 [(0, ''), (4, '('), (10, 'E'), (6, '+'), (5, 'n')]
20 5,n -- )
21 F->n
22 [(0, ''), (4, '('), (10, 'E'), (6, '+'), (3, 'F')]
23 3,F -- )
24 T->F
25 [(0, ''), (4, '('), (10, 'E'), (6, '+'), (11, 'T')]
26 11,T -- )
27 E->E+T
28 [(0, ''), (4, '('), (10, 'E')]
29 10,E -- )
30 [(0, ''), (4, '('), (10, 'E'), (15, ')')]
31 15,) -- *
32 F->(E)
33 [(0, ''), (3, 'F')]
34 3,F -- *
35 T->F
36 [(0, ''), (2, 'T')]
37 2,T -- *
38 [(0, ''), (2, 'T'), (8, '*')]
39 8,* -- n
40 [(0, ''), (2, 'T'), (8, '*'), (5, 'n')]
41 5,n -- -
42 F->n
43 [(0, ''), (2, 'T'), (8, '*'), (13, 'F')]
44 13,F -- -
45 T->T*F
46 [(0, ''), (2, 'T')]
47 2,T -- -
48 E->T
49 [(0, ''), (1, 'E')]
50 1,E -- -
51 [(0, ''), (1, 'E'), (7, '-')]
52 7,- -- n
53 [(0, ''), (1, 'E'), (7, '-'), (5, 'n')]
54 5,n -- $
55 F->n
56 [(0, ''), (1, 'E'), (7, '-'), (3, 'F')]

```

```
57 3,F -- $
58 T->F
59 [(0, '('), (1, 'E'), (7, '-'), (12, 'T')]
60 12,T -- $
61 E->E-T
62 [(0, '('), (1, 'E')]
63 1,E -- $
64 S->E
```

5 实验总结

在这次实验中，我实现了 $LL(1)$ 以及 $SLR(1)$ 进行表达式的语法解析。通过这次实验，我对课本上的理论知识有了更深刻的理解。