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：

```
class Lexer:
    def __init__(self, expression):
        self.token = None
        self.expression = (expression + '$').replace("␣", "")

    def next_token(self):
        if self.token:
            temp = self.token
            self.token = None
            return temp
        else:
            return self._next_token()

    def peek_token(self):
        if not self.token:
            self.token = self._next_token()
        return self.token

    def _next_token(self):
        if self.expression == '':
            return None
        c = self.expression[0]
        if c == '(' or c == ')' or c == '+' or c == '-' or c == '*'
           or c == '/' or c == '$':
            self.expression = self.expression[1:]
            return c
        else:
            result = None
```

```python
            i = 1
            try:
                while i < len(self.expression):
                    if self.expression[i - 1] == '.':
                        i += 1
                    result = float(self.expression[:i])
                    i += 1
            except:
                self.expression = self.expression[i - 1:]
                return 'n'#result
        t = len(self.expression)
        self.expression = self.expression[i - 1:]
        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 $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(α), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ 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 $E \to E + T \mid T$ 转化为 $E \to TE'$ 以及 $E' \to +TE' \mid \epsilon$ 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：

```python
class Grammer:
    def __init__(self):
        self.startSymbol = 'S' # default start symbol
        self.nonTerminalSymbol = set()
        self.terminalSymbol = set()
        self.generatorExpression = {}

        # Temp variable
        self.nullable = {}
        self.firstSymbols = {}
        self.followSymbols = {}
        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：

```python
class LLParser:
    def __init__(self, grammer,expression):
        self.grammer = grammer
        self.lexer = Lexer(expression)
        self.stack = ['$', self.grammer.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：

```python
class Grammer:
    def __init__(self):
        self.startSymbol = 'S'  # default start symbol
        self.nonTerminalSymbol = set()
        self.terminalSymbol = set()
        self.generatorExpression = {}

        # Temp variable
        self.nullable = {}
        self.firstSymbols = {}
        self.followSymbols = {}
        self.llTable = {}

    def get_unused_non_terminal_symbol(self):
        tmp = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
        for i in tmp:
            if i not in self.nonTerminalSymbol:
                self.nonTerminalSymbol.add(i)
                return i

    def solve_left_recursion(self):
        keys = list(self.generatorExpression.keys())
        for k in keys:
            v = self.generatorExpression[k]
            leftRecu = [i for i in v if i[0] == k]
            if leftRecu:
                newSymbol = self.get_unused_non_terminal_symbol()
                self.generatorExpression[k] = [
                    i + newSymbol for i in v if i[0] != k]
                self.generatorExpression[newSymbol] = [
                    i[1:] + newSymbol for i in leftRecu]
                self.generatorExpression[newSymbol].append('')

    def compute_first_and_follow_set(self):
        old_nullable = {}
        for k in self.nonTerminalSymbol:
```

```
            self.nullable[k] = False
        while old_nullable != self.nullable:
            old_nullable = self.nullable.copy()
            for k, v in self.generatorExpression.items():
                self.nullable[k] = False
                for expressions in v:
                    nullable = True
                    for i in expressions:
                        if not (i in self.nonTerminalSymbol and '' in self.generatorExpression[i]):
                            nullable = False
                    self.nullable[k] = self.nullable[k] or nullable

        old_first_set = set()
        for k in self.nonTerminalSymbol:
            self.firstSymbols[k] = set()
        for k, v in self.generatorExpression.items():
            for expressions in v:
                for i in expressions:
                    if i not in self.nonTerminalSymbol:
                        self.firstSymbols[i] = set(i)
        while(old_first_set != self.firstSymbols):
            old_first_set = self.firstSymbols.copy()
            for k, v in self.generatorExpression.items():
                for expressions in v:
                    add_new_first = True
                    if expressions == '':
                        self.firstSymbols[k].add('')
                    for i in expressions:
                        if add_new_first:
                            self.firstSymbols[k] = self.firstSymbols[k].union(
                                self.firstSymbols[i])
                            add_new_first = add_new_first and (
                                i in self.nonTerminalSymbol and self.nullable[i])

        old_follow_set = set()
        for k in self.nonTerminalSymbol:
```

```
            self.followSymbols[k] = set()
        for k, v in self.generatorExpression.items():
            for expressions in v:
                for i in expressions:
                    if i not in self.nonTerminalSymbol:
                        self.followSymbols[i] = set(i)
        self.followSymbols[self.startSymbol].add('$')
        while(old_follow_set != self.followSymbols):
            old_follow_set = self.followSymbols.copy()
            # print(self.generatorExpression.items())
            for key, v in self.generatorExpression.items():
                for expressions in v:
                    for i in range(len(expressions)):
                        add_follow = True
                        for j in range(i + 1, len(expressions)):
                            add_follow = add_follow and (
                                expressions[j] in self.
                                   nonTerminalSymbol and self.
                                   nullable[expressions[j]])
                        if add_follow:
                            self.followSymbols[
                                expressions[i]] = self.followSymbols[
                                    expressions[i]].union(
                                        self.followSymbols[key])
                        for j in range(i + 1, len(expressions)):
                            add_follow = True
                            for k in range(i + 1, j):
                                add_follow = add_follow and (
                                    expressions[k] in self.
                                       nonTerminalSymbol and self.
                                       nullable[expressions[k]])
                            if add_follow:
                                self.followSymbols[expressions[i]] =
                                   self.followSymbols[expressions[i
                                   ]].union(
                                    self.firstSymbols[expressions[j
                                       ]])
        keys = list(self.firstSymbols.keys())
        for k in keys:
```

```
                if k not in self.nonTerminalSymbol:
                    self.firstSymbols.pop(k)
            keys = list(self.followSymbols.keys())
            for k in keys:
                if k not in self.nonTerminalSymbol:
                    self.followSymbols.pop(k)
            for k, v in self.followSymbols.items():
                if '' in v:
                    v.remove('')

        def _get_first(self, expression):
            if expression == '':
                return set('')
            result = set()
            for i in expression:
                if i in self.nonTerminalSymbol:
                    result = result.union(self.firstSymbols[i])
                    if not self.nullable[i]:
                        return result
                else:
                    return result.union(i)
            return result

        def get_first(self, expression):
            result = self._get_first(expression)
            if '' in result:
                result.remove('')
            return result

        def get_nullable(self, expression):
            if expression == '':
                return True
            return '' in self._get_first(expression)

        def build_ll_table(self):
            for k in self.nonTerminalSymbol:
                self.llTable[k] = {}
            for k, v in self.generatorExpression.items():
                for expressions in v:
```

```
                for i in self.get_first(expressions):
                    self.llTable[k][i] = expressions
                if self.get_nullable(expressions):
                    for i in self.followSymbols[k]:
                        self.llTable[k][i] = expressions


class Lexer:
    def __init__(self, expression):
        self.token = None
        self.expression = (expression + '$').replace(" ", "")

    def next_token(self):
        if self.token:
            temp = self.token
            self.token = None
            return temp
        else:
            return self._next_token()

    def peek_token(self):
        if not self.token:
            self.token = self._next_token()
        return self.token

    def _next_token(self):
        if self.expression == '':
            return None
        c = self.expression[0]
        if c == '(' or c == ')' or c == '+' or c == '-' or c == '*'
           or c == '/' or c == '$':
            self.expression = self.expression[1:]
            return c
        else:
            result = None
            i = 1
            try:
                while i < len(self.expression):
                    if self.expression[i - 1] == '.':
```

```
                        i += 1
                    result = float(self.expression[:i])
                    i += 1
            except:
                self.expression = self.expression[i - 1:]
                return 'n'  # result
        t = len(self.expression)
        self.expression = self.expression[i - 1:]
        return 'n' if i == t else result


class LLParser:
    def __init__(self, grammer, expression):
        self.grammer = grammer
        self.lexer = Lexer(expression)
        self.stack = ['$', self.grammer.startSymbol]

    def _parse(self):
        if self.stack[-1] not in self.grammer.nonTerminalSymbol:
            if self.stack[-1] == self.lexer.peek_token():
                self.stack.pop()
                self.lexer.next_token()
            else:
                raise RuntimeError
        else:
            expression = self.grammer.llTable[self.stack[-1]
                                              ][self.lexer.peek_token()]
            print(self.stack[-1] + ' -> ' +
                  ('\'\'' if expression == '' else expression))
            self.stack.pop()
            for x in expression[::-1]:
                self.stack.append(x)

    def parse(self):
        while self.stack[-1] != '$':
            self._parse()
```

```python
if __name__ == '__main__':
    expression = "(1 + 2) + 2.5"
    grammer = Grammer()
    grammer.startSymbol = 'E'
    grammer.nonTerminalSymbol.add('E')
    grammer.generatorExpression['E'] = ['E+T', 'E-T', 'T']
    grammer.nonTerminalSymbol.add('T')
    grammer.generatorExpression['T'] = ['T*F', 'T/F', 'F']
    grammer.nonTerminalSymbol.add('F')
    grammer.terminalSymbol.add('n')
    grammer.generatorExpression['F'] = ['(E)', 'n']
    print(grammer.generatorExpression)
    grammer.solve_left_recursion()
    print(grammer.generatorExpression)
    grammer.compute_first_and_follow_set()
    print(grammer.nullable)
    print(grammer.firstSymbols)
    print(grammer.followSymbols)
    grammer.build_ll_table()
    print(grammer.llTable)
    ll = LLParser(grammer, expression)
    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：

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

```
': 'FB', '(': 'FB'}, 'A': {'+': '+TA', '-': '-TA', ')': '', '$': ''}}
E -> TA
T -> FB
F -> (E)
E -> TA
T -> FB
F -> n
B -> ''
A -> +TA
T -> FB
F -> n
B -> ''
A -> ''
B -> ''
A -> +TA
T -> FB
F -> n
B -> ''
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。

(0) $S \to E$ (1) $E \to E+T$ (2) $E \to E-T$

(3) $E \to T$ (4) $T \to T*F$ (5) $T \to T/F$

(6) $T \to F$ (7) $F \to (E)$ (8) $F \to id$

	+	-	*	/	(	)	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										—

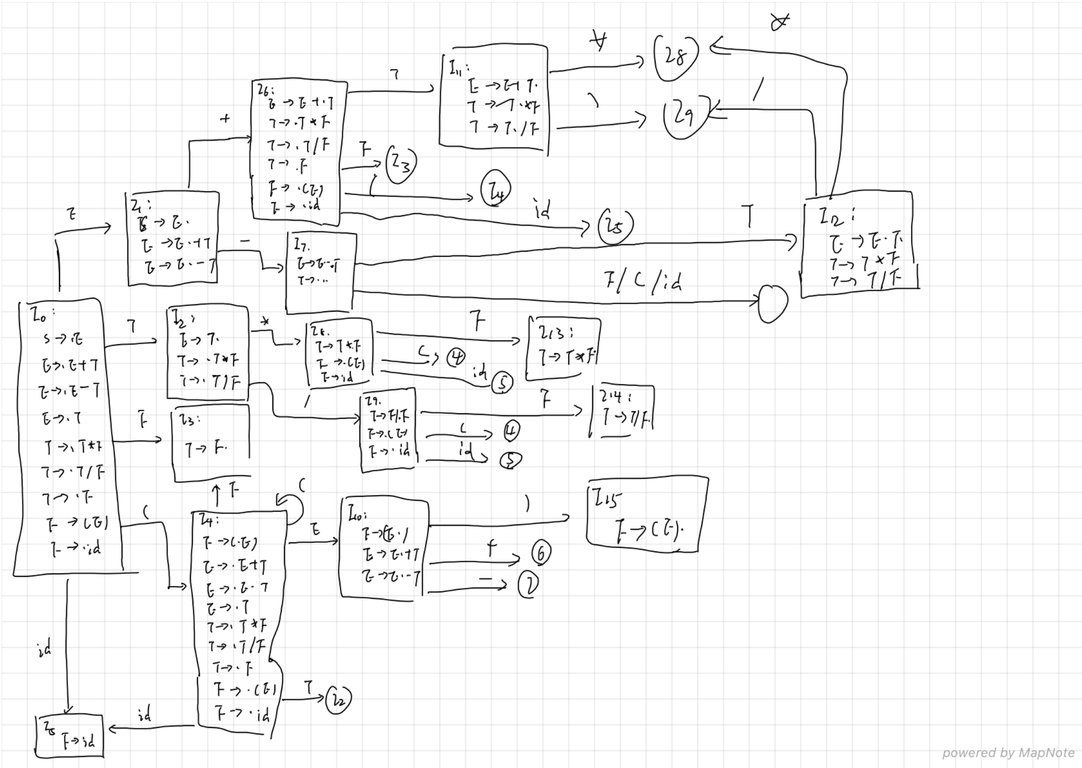

图 2: 构造 DFA

4.2 实验代码

解析算法与书上一致。主要代码如下：

```
class Lexer:
    def __init__(self, expression):
        self.token = None
        self.expression = (expression + '$').replace("␣", "")

    def next_token(self):
        if self.token:
            temp = self.token
            self.token = None
            return temp
        else:
            return self._next_token()

    def peek_token(self):
        if not self.token:
            self.token = self._next_token()
        return self.token
```

```python

    def _next_token(self):
        if self.expression == '':
            return None
        c = self.expression[0]
        if c == '(' or c == ')' or c == '+' or c == '-' or c == '*' or c == '/' or c == '$':
            self.expression = self.expression[1:]
            return c
        else:
            result = None
            i = 1
            try:
                while i < len(self.expression):
                    if self.expression[i - 1] == '.':
                        i += 1
                    result = float(self.expression[:i])
                    i += 1
            except:
                self.expression = self.expression[i - 1:]
                return 'n'  # result
        t = len(self.expression)
        self.expression = self.expression[i - 1:]
        return 'n' if i == t else result


class LR1Table:
    def __init__(self):
        self.grammer = {}
        self.action = {}
        self.goto = {}


class LR1Parser:
    def __init__(self, table, expression):
        self.stack = []
        self.table = table
        self.lexer = Lexer(expression)
        self.stack.append((0, ''))
```

```

    def _parse(self):
        print(self.stack)
        s = self.stack[-1][0]
        a = self.lexer.peek_token()
        print(str(s) + ',' + self.stack[-1][1] + ' -- ' + a)
        action = self.table.action[s][a]
        if action[0] == 'S':
            self.stack.append((action[1], a))
            self.lexer.next_token()
            return True
        elif action[0] == 'R':
            r_expression = self.table.grammer[action[1]]
            for i in range(len(r_expression[1])):
                self.stack.pop()
            if action[1] != 0:
                self.stack.append((
                    self.table.goto[self.stack[-1][0]][r_expression[0]], r_expression[0]))
            print(r_expression[0] + '->' +
                  r_expression[1])
            return action[1] != 0
        else:
            print('Error in parsing')
            return False

    def parse(self):
        result = True
        while (result):
            result = self._parse()


if __name__ == '__main__':
    table = LR1Table()
    table.grammer = {
        0: ("S", "E"),
        1: ("E", "E+T"),
        2: ("E", "E-T"),
        3: ("E", "T"),
```

```
        4: ("T", "T*F"),
        5: ("T", "T/F"),
        6: ("T", "F"),
        7: ("F", "(E)"),
        8: ("F", "n")
    }
    table.goto = {
        0: {
            'E': 1,
            'T': 2,
            'F': 3
        },
        4: {
            'E': 10,
            'T': 2,
            'F': 3
        },
        6: {
            'T': 11,
            'F': 3
        },
        7: {
            'T': 12,
            'F': 3
        },
        8: {
            'F': 13
        },
        9: {
            'F': 14
        }
    }
    table.action = {
        0: {
            '(': ('S', 4),
            'n': ('S', 5)
        },
        1: {
            '+': ('S', 6),
```

```
            '-': ('S', 7),
            '$': ('R', 0)
        },
        2: {
            '+': ('R', 3),
            '-': ('R', 3),
            '*': ('S', 8),
            '/': ('S', 9),
            ')': ('R', 3),
            '$': ('R', 3)
        },
        3: {
            '+': ('R', 6),
            '-': ('R', 6),
            '*': ('R', 6),
            '/': ('R', 6),
            ')': ('R', 6),
            '$': ('R', 6)
        },
        4: {
            '(': ('S', 4),
            'n': ('S', 5)
        },
        5: {
            '+': ('R', 8),
            '-': ('R', 8),
            '*': ('R', 8),
            '/': ('R', 8),
            ')': ('R', 8),
            '$': ('R', 8)
        },
        6: {
            '(': ('S', 4),
            'n': ('S', 5)
        },
        7: {
            '(': ('S', 4),
            'n': ('S', 5)
        },
```

```
        8: {
            '(': ('S', 4),
            'n': ('S', 5)
        },
        9: {
            '(': ('S', 4),
            'n': ('S', 5)
        },
        10: {
            '+': ('S', 6),
            '-': ('S', 7),
            ')': ('S', 15)
        },
        11: {
            '+': ('R', 1),
            '-': ('R', 1),
            '*': ('S', 8),
            '/': ('S', 9),
            ')': ('R', 1),
            '$': ('R', 1)
        },
        12: {
            '+': ('R', 2),
            '-': ('R', 2),
            '*': ('S', 8),
            '/': ('S', 9),
            ')': ('R', 2),
            '$': ('R', 2)
        },
        13: {
            '+': ('R', 4),
            '-': ('R', 4),
            '*': ('R', 4),
            '/': ('R', 4),
            ')': ('R', 4),
            '$': ('R', 4)
        },
        14: {
            '+': ('R', 5),
```

```
                '-': ('R', 5),
                '*': ('R', 5),
                '/': ('R', 5),
                ')': ('R', 5),
                '$': ('R', 5)
            },
            15: {
                '+': ('R', 7),
                '-': ('R', 7),
                '*': ('R', 7),
                '/': ('R', 7),
                ')': ('R', 7),
                '$': ('R', 7)
            }
        }
    parser = LR1Parser(table, "(1 + 2) * 3 - 4")
    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：

```
PS D:\playground\FuckingCalculator> python .\LR1\LR1.py
[(0, '')]
0, -- (
[(0, ''), (4, '(')]
4,( -- n
[(0, ''), (4, '('), (5, 'n')]
5,n -- +
F->n
[(0, ''), (4, '('), (3, 'F')]
3,F -- +
T->F
[(0, ''), (4, '('), (2, 'T')]
2,T -- +
E->T
[(0, ''), (4, '('), (10, 'E')]
10,E -- +
[(0, ''), (4, '('), (10, 'E'), (6, '+')]
```

```
6,+ -- n
[(0, ''), (4, '('), (10, 'E'), (6, '+'), (5, 'n')]
5,n -- )
F->n
[(0, ''), (4, '('), (10, 'E'), (6, '+'), (3, 'F')]
3,F -- )
T->F
[(0, ''), (4, '('), (10, 'E'), (6, '+'), (11, 'T')]
11,T -- )
E->E+T
[(0, ''), (4, '('), (10, 'E')]
10,E -- )
[(0, ''), (4, '('), (10, 'E'), (15, ')')]
15,) -- *
F->(E)
[(0, ''), (3, 'F')]
3,F -- *
T->F
[(0, ''), (2, 'T')]
2,T -- *
[(0, ''), (2, 'T'), (8, '*')]
8,* -- n
[(0, ''), (2, 'T'), (8, '*'), (5, 'n')]
5,n -- -
F->n
[(0, ''), (2, 'T'), (8, '*'), (13, 'F')]
13,F -- -
T->T*F
[(0, ''), (2, 'T')]
2,T -- -
E->T
[(0, ''), (1, 'E')]
1,E -- -
[(0, ''), (1, 'E'), (7, '-')]
7,- -- n
[(0, ''), (1, 'E'), (7, '-'), (5, 'n')]
5,n -- $
F->n
[(0, ''), (1, 'E'), (7, '-'), (3, 'F')]
```

```
3,F -- $
T->F
[(0, ''), (1, 'E'), (7, '-'), (12, 'T')]
12,T -- $
E->E-T
[(0, ''), (1, 'E')]
1,E -- $
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，我对课本上的理论内容有了更深刻的理解。