自底向上语法分析程序的设计与实现

班级：2021211307
学号：2021211153
姓名：谭若樱

1 实验任务

对于以下文法：

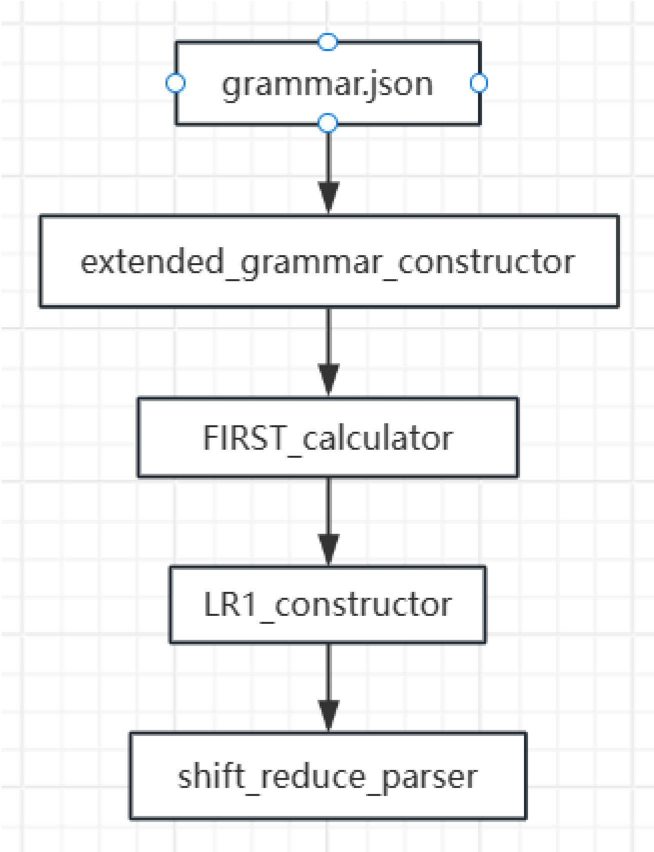
```
E → E + T | E - T | T
T → T * F | T / F | F
F → (E) | num
```

编写自底向上的语法分析程序实现以下功能：

- 构造识别该文法的所有活前缀的DFA
- 构造该文法的LR分析表
- 对于给定的算数表达式，根据LR分析表进行语法分析

2 模块介绍

各模块及其关系图如下：



其中：

- $grammar.json$为给定文法以$JSON$格式存储的文件
- $extended_grammar_constructor$实现了将某文法改写成拓广文法
- $FIRST_calculator$实现了计算某文法中所有文法符号的FIRST集合
- $LR1_constructor$实现了计算某拓广文法的LR(1)分析表
- $shift_reduce_parser$实现了根据命令行输入与给定LR(1)分析表实现移入-归约分析，并将分析过程通过图形化窗口显示

2.1 文法记法

对于某文法，通过$terminal$、$non\_terminal$、$start\_symbol$、$production$进行描述：

- $terminal$ ：表示该文法的终结符
- $non\_terminal$ ：表示该文法的非终结符
- $start\_symbol$ ：表示该文法的起始符
- $production$ ：表示该文法的产生式

对于题目给定的文法，根据以上描述方式通过$JSON$格式进行存储，存储内容如下：

```json
{
    "terminal": [
        "+",
        "-",
        "*",
        "/",
        "(",
        ")",
        "num"
    ],
    "non_terminal": [
        "E",
        "T",
        "F"
    ],
    "start_symbol": "E",
    "production": [
        "E -> E + T",
        "E -> E - T",
        "E -> T",
        "T -> T * F",
        "T -> T / F",
        "T -> F",
        "F -> ( E )",
        "F -> num"
    ]
}
```

其中:

- $terminal, non\_terminal, production$ 均为 $list$ 类型
- $start\_symbol$ 为 $string$ 类型

以上JSON存储格式内容为以下代码生成:

```python
# 该模块用于输入待分析的文法

import json

# 文法字典
grammar = {}

terminal = ['+', '-', '*', '/', '(', ')', 'num']     # 终结符
non_terminal = ['E', 'T', 'F']                       # 非终结符
start_symbol = 'E'                                   # 起始符
production = []                                      # 产生式

# 加入产生式
production.append('E -> E + T')
production.append('E -> E - T')
production.append('E -> T')
production.append('T -> T * F')
production.append('T -> T / F')
production.append('T -> F')
production.append('F -> ( E )')
production.append('F -> num')

# 将文法写入文法字典
grammar['terminal'] = terminal
grammar['non_terminal'] = non_terminal
grammar['start_symbol'] = start_symbol
grammar['production'] = production

# 把文法字典写入文件grammar.json
with open("grammar.json", 'w') as json_file:
    json.dump(grammar, json_file, indent=4)
```

2.2 拓广文法改写

为了确保在LR(1)分析表中只有一个起始状态，需要将原始文法改写成拓广文法。

拓广文法是对原始文法进行扩展，添加一个新的起始符号和一个新的产生式，即：对于给定的文法 $G=(N,T,P,E)$，生成等价的拓广文法 $G'=(N\cup E',T,P\cup E'\to E,E')$

具体的实现方式如下：

```python
# 返回改写后拓广文法的字典
def constructor(grammar_file: str) -> dict:
    # 读入输入的文法
    with open(grammar_file, 'r') as json_file:
        grammar = json.load(json_file)

    # 起始符改为 E'
    start_symbol = 'E\''

    # 将新的起始符 E' 加入非终结符集合中
    non_terminal = grammar.get('non_terminal', None)
    non_terminal.insert(0, 'E\'')

    # 加入一条产生式 E' -> E
    production = grammar.get('production', None)
    production.insert(0, 'E\' -> E')

    grammar.update({'start_symbol' : start_symbol,
                    'non_terminal' : non_terminal,
                    'production':production})
    return grammar
```

在$extended_grammar_constructor.py$脚本中加入：

```python
if __name__ == "__main__":
    grammar = constructor('grammar.json')
    # 把拓广文法字典写入文件extended_grammar.json
    with open("extended_grammar.json", 'w') as json_file:
        json.dump(grammar, json_file, indent=4)
```

运行$extended_grammar_constructor.py$脚本，可以看到该模块给出的改写后的拓广文法为：

```
{
    "terminal": [
        "+",
        "-",
        "*",
        "/",
        "(",
        ")",
        "num"
    ],
    "non_terminal": [
        "E'",
        "E",
        "T",
        "F"
    ],
    "start_symbol": "E'",
    "production": [
        "E' -> E",
        "E -> E + T",
        "E -> E - T",
        "E -> T",
        "T -> T * F",
        "T -> T / F",
        "T -> F",
        "F -> ( E )",
        "F -> num"
    ]
}
```

2.3 构建非终结符和待约项目的 FIRST 集

对于任意产生式 $A \to \alpha$ ，若 $\alpha \neq \varepsilon$，设该产生式为：

$$A \to Y_1 Y_2 ... Y_k$$

遍历产生式右部的每一个 Y_i，如果：

- Y_i 是终结符，则 α 的 $FIRST$ 集中增加 Y_i，终止遍历；
- Y_i 是非终结符，则将它的 $FIRST$ 集中非 ε 元素加入 A 的 $FIRST$ 集中。此后检查 Y_i 的 $FIRST$ 集中是否包含 ε ，若不包含，则终止遍历。

特别的，若：$\alpha \to \varepsilon$，则将 ε 加入 A 的 $FIRST$ 集中。

具体的实现如下：

```python
# 将给定产生式的左部和右部分开
def get_symbol(production: str) -> Tuple[str, list]:
    left_part, right_part = production.split('->')
    left_part = left_part.strip()
    right_symbols = right_part.strip().split(' ')
    return left_part, right_symbols

# 根据给定的FIRST集计算当前string的FIRST集
def get_string_FIRST(string: list, FIRST: dict) -> list:
    res = []
    empty_stringable = True
    for symbol in string:
        if 'ε' not in FIRST.get(symbol):
            res.extend(FIRST.get(symbol))
            empty_stringable = False
            break
        else:
            # 加入除 ε 外所有元素
            res.extend(x for x in FIRST.get(symbol) if x != 'ε')
    # 所有文法符号均可为空串，则FIRST中包含ε
    if empty_stringable:
        res.append('ε')
    # 去重
    return list(set(res))

# 计算给定文法各个符号的FIRST
def constructor(grammar: dict) -> dict:
    # 文法符号的FIRST集的字典，映射关系为文法符号->FIRST集合
    FIRST = {}

    # 对终结符 a 有 FIRST(a) = {a}
    terminal = grammar.get('terminal', None)
    for symbol in terminal:
        FIRST[symbol] = [symbol]

    # 构造非终结符的FIRST集
    non_terminal = grammar.get('non_terminal', None)
    for symbol in non_terminal:
        FIRST[symbol] = []
    production = grammar.get('production', None)
    while True:
        modified = False
        for p in production:
```

```python
            left, right = get_symbol(p)
            # print(left, right)
            if right == ['ε']: # 产生式形如 A -> ε
                if 'ε' not in FIRST.get(left):
                    FIRST.get(left).append('ε')
                    modified = True
            else:
                added_symbols = get_string_FIRST(right, FIRST)
                for symbol in added_symbols:
                    if symbol not in FIRST.get(left):
                        FIRST.get(left).append(symbol)
                        modified = True
        if modified == False:
            break
    return FIRST
```

在$FIRST_calculator.py$脚本中加入：

```python
if __name__ == "__main__":
    grammar = EGC('grammar.json')
    FIRST = constructor(grammar)
    # 把FIRST集写入文件FIRST.json
    with open("FIRST.json", 'w') as json_file:
        json.dump(FIRST, json_file, indent=4)
```

运行$FIRST_calculator.py$脚本，可以看到该模块计算得到各个文法符号的$FIRST$集为：

```
{
    "+": [
        "+"
    ],
    "-": [
        "-"
    ],
    "*": [
        "*"
    ],
    "/": [
        "/"
    ],
    "(": [
        "("
    ],
    ")": [
        ")"
    ],
    "num": [
        "num"
    ],
    "E'": [
        "(",
        "num"
    ],
    "E": [
        "(",
        "num"
    ],
    "T": [
        "(",
        "num"
    ],
    "F": [
        "(",
        "num"
    ]
}
```

2.4 LR(1)分析表构造

构造 $LR(1)$ 项目集的闭包，构造过程如下：

- 初始化 $closure(I) \leftarrow I$
- 对于 $[A \rightarrow \alpha \cdot B\beta, a] \in closure(I)$，若 $B \rightarrow \eta \in P$，则 $\forall b \in FIRST(\beta a)$，使 $closure(I) \leftarrow closure(I) \cup [B \rightarrow \cdot \eta, b]$
- 重复以上过程直至 $closure(I)$ 不再增大为止

具体实现如下：

```python
# 计算项目集闭包
def closure(item: list, grammar: dict, FIRST: dict) -> list:
    '''
    item = [[production, dot, lookahead]]
    其中 dot 表示点在产生式右部第几个文法符号的前面(从0开始编号),
    特别的, 当 dot = len(右部) 时表示规约项目
    '''
    grammar_production = grammar.get('production')
    grammar_left = grammar.get('left')
    grammar_right = grammar.get('right')
    grammar_non_terminal = grammar.get('non_terminal')
    count = 0
    while count < len(item):
        i = item[count]
        cur_production_index = grammar_production.index(i[0]) # 当前要处理的产生式的索引
        if i[1] == len(grammar_right[cur_production_index]): # 是一个归约项目
            pass
        elif grammar_right[cur_production_index][i[1]] in grammar_non_terminal: # 是一个待约项目
            # 对于形如 A -> α · B, 找到所有 B 为左部的产生式的 index
            indices = []
            for index, value in enumerate(grammar_left):
                if value == grammar_right[cur_production_index][i[1]]:
                    indices.append(index)
            # 计算向前看符号串
            r = grammar_right[cur_production_index][(i[1] + 1):]
            r.append(i[2])
            lookahead = get_string_FIRST(r, FIRST)
            new_item = []
            for index in indices:
                for symbol in lookahead:
                    new_item.append([grammar_production[index], 0, symbol])
            for it in new_item:
                if it not in item:
                    item.append(it)
        else: # 是一个移进项目
            pass
        count += 1
    return item
```

构造 $LR(1)$ 项目集规范族并构建 $LR(1)$ 分析表，构造过程如下：

- 初始化项目集规范族$items \leftarrow closure(E \rightarrow E', \$)$

- 遍历项目集规范族，对里面每一个项目集遍历每一个项目$item$：
  - 若该项目是一个形如$[A \rightarrow \alpha \cdot a\beta, b]$的移进项目，则找到项目集中所有形如$[A \rightarrow \alpha' \cdot a\beta', b']$的项目，将$[A \rightarrow \alpha' a \cdot \beta', b']$加入新的项目集$new_item$。
    - 若$closure(new_item) \in items$,则令$LR1[a] = ['S', items.index(closure(new_item))]$;
    - 否则$LR1[a] = ['S', len(items)]$
    - 该过程同时检查是否存在移入-归约冲突，存在冲突则直接返回$None$
  - 若该项目是一个形如$[A \rightarrow \alpha \cdot B\beta, a]$的待约项目，则找到项目集中所有形如$[A' \rightarrow \alpha' \cdot B\beta', a']$的项目，将$\forall a' \in FIRST(\beta' a'), [A' \rightarrow \alpha' \cdot B\beta', a']$加入新的项目集$new_item$
    - 若$closure(new_item) \in items$,则令$LR1['B'] = items.index(closure(new_item))$;
    - 否则$LR1['B'] = len(items)$
  - 若该项目是一个归约项目[A → B ·, a]，则:
    - 若该项目为$[E' \rightarrow E \cdot, \$]$，则$LR1[items.index(item)]['\$'] =' ACC'$
    - 否则$LR1[items.index(item)]['a'] = ['R', production_index]$

具体实现如下：

```python
# 构造该文法的LR(1)项目集规范族
def constructor(grammar: dict, FIRST: dict) -> dict:
    # 将产生式拆成左部和右部，方便后续操作
    production = grammar.get('production', None)
    left, right = split_production(production)
    grammar['left'] = left
    grammar['right'] = right
    terminal = grammar.get('terminal', None)
    LR1 = {} # LR(1)分析表
    '''
    LR1 = {state1 : {'A' : state2, 'a' : ['R'/'S', index]}}
    表示在state1遇到非终结符转移到state2,
    遇到终结符a根据第index条产生式规约或
    移入并转移到状态index
    '''
    items = [] # 项目集规范族
    equal_item = {}
    items.append(closure([["E' -> E", 0, '$']], grammar, FIRST))
    state_index = 0
    while state_index < len(items):
        cur_item = items[state_index]
        # 计算状态state_index的goto和action
        LR1[state_index] = {}
        is_checked = [False] * len(cur_item)
        for i, item in enumerate(cur_item):
            if is_checked[i]: # 已经加入新的项目了
                continue
            else:
                production_index = production.index(item[0])
                condition_1 = item[1] == len(right[production_index])
                condition_2 = right[production_index] == ['ε']
                if condition_1 or condition_2: # 是一个归约项目
                    if LR1[state_index].get(item[2], None) == None:
                        if production_index == 0:
                            LR1[state_index][item[2]] = 'ACC'
                        else:
                            LR1[state_index][item[2]] = ['R', production_index]
                    elif LR1[state_index][item[2]] != ['R', production_index]:
                        print('归约存在冲突')
                        return None # 存在冲突，不是LR(1)文法，返回None
                elif right[production_index][item[1]] in terminal: # 是一个移进项目
                    # 对于 a ，找到项目集中所有形如A -> α · a β的项目
                    indices = get_same_items_index(item, cur_item, production, right)
```

```python
                new_item = []
                for index in indices:
                    is_checked[index] = True
                    new_item.append([cur_item[index][0],
                                     cur_item[index][1] + 1,
                                     cur_item[index][2]])
                new_item = closure(new_item, grammar, FIRST)
                new_item_index = get_item_index(new_item, items)
                symbol = right[production_index][item[1]]
                if LR1[state_index].get(symbol, None) == None:
                    LR1[state_index][symbol] = ['S', new_item_index]
                elif LR1[state_index][symbol] != ['S', new_item_index]:
                    print('移进项目存在冲突')
                    return None
                if new_item_index == len(items):
                    items.append(new_item)
            else: # 是一个待约项目
                # 对于 B ,找到项目集中所有形如A -> α · B β的项目
                indices = get_same_items_index(item, cur_item, production, right)
                new_item = []
                for index in indices:
                    is_checked[index] = True
                    new_item.append([cur_item[index][0],
                                     cur_item[index][1] + 1,
                                     cur_item[index][2]])
                new_item = closure(new_item, grammar, FIRST)
                new_item_index = get_item_index(new_item, items)
                symbol = right[production_index][item[1]]
                if LR1[state_index].get(symbol, None) == None:
                    LR1[state_index][symbol] = new_item_index
                elif LR1[state_index][symbol] != new_item_index:
                    print('待约项目存在冲突')
                    return None
                if new_item_index == len(items):
                    items.append(new_item)
        state_index += 1
    with open("closure.json", 'w') as json_file:
        json.dump(items, json_file, indent=2)
    return LR1
```

2.5 LR(1)分析

LR(1)分析过程伪代码如下

```
Function LR1Parser(input_string):
    statestack.push(0)
    symbolstack.push(NULL)
    buffer ← input_string + '$'
    ip ← 0
    answer ← []

    while True:
        X ← statestack.top()
        a ← buffer[ip]

        if action[X, a] = Shift S':
            statestack.push(S')
            symbolstack.push(a)
            ip ← ip + 1
        else if action[X, a] = Reduce A → β:
            reduction ← A → β
            for i = 1 to |β| do
                statestack.pop()
                symbolstack.pop()

            X' ← statestack.top()
            statestack.push(goto[X', A])
            symbolstack.push(A)
            answer.push_back(reduction)
        else if action[X, a] = ACC:
            return answer
        else:
            return error
```

具体实现代码如下，根据$PyQt5$框架实现简单的UI界面：

```python
State = [0]                         # 状态栈
Symble = ['-']                      # 符号栈
user_input = input()                # 用户输入
user_input = user_input.split()
input_queue = deque()
for word in user_input:
    input_queue.append(word)
input_queue.append('$')
output = None

class LR1Parsing(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        global grammar, non_terminal, terminal, production
        global FIRST, LR1, State, Symble, input_queue, output
        self.setGeometry(100, 100, 800, 800)
        self.tableWidget = QTableWidget(self)
        self.tableWidget.setGeometry(0, 0, 800, 800)
        self.tableWidget.setColumnCount(4)
        self.tableWidget.setColumnWidth(0, 100)
        self.tableWidget.setColumnWidth(1, 100)
        self.tableWidget.setColumnWidth(2, 200)
        self.tableWidget.setColumnWidth(3, 400)
        self.tableWidget.setHorizontalHeaderLabels(['State栈', 'Symble栈', '输入', '分析动作'])
        print('start parsing')
        self.startParsing()
    def addTableRow(self, State, Symble, input_queue, output):
        rowPosition = self.tableWidget.rowCount()
        self.tableWidget.insertRow(rowPosition)
        state = ' '.join(map(str, State))
        symble = ' '.join(Symble)
        cur_input = ' '.join(input_queue)
        cur_output = ' '.join(output)
        self.tableWidget.setItem(rowPosition, 0, QTableWidgetItem(state))
        self.tableWidget.setItem(rowPosition, 1, QTableWidgetItem(symble))
        self.tableWidget.setItem(rowPosition, 2, QTableWidgetItem(cur_input))
        self.tableWidget.setItem(rowPosition, 3, QTableWidgetItem(cur_output))
    def startParsing(self):
        while True:
            if len(State) == len(Symble):
                cur_state = State[-1]
```

```python
            state_transform = LR1.get(cur_state, None)
            input_stack_top = input_queue[0]
            ACTION = state_transform.get(input_stack_top, None)
            if ACTION == None:
                output = 'Error'
                self.addTableRow(State, Symble, input_queue, output)
                break
            elif ACTION == 'ACC':
                output = 'ACC'
                self.addTableRow(State, Symble, input_queue, output)
                break
            elif ACTION[0] == 'S':
                output = 'Shift ' + str(ACTION[1])
                self.addTableRow(State, Symble, input_queue, output)
                input_queue.popleft()
                State.append(ACTION[1])
                Symble.append(input_stack_top)
            else:
                output = 'Reduce by ' + production[ACTION[1]]
                self.addTableRow(State, Symble, input_queue, output)
                left_part, right_part = production[ACTION[1]].split('->')
                left_part = left_part.strip()
                right_symbols = right_part.strip().split(' ')
                for symbol in reversed(right_symbols):
                    if symbol == Symble[-1]:
                        State.pop()
                        Symble.pop()
                    else:
                        output = 'Error'
                        self.addTableRow(State, Symble, input_queue, output)
                        break
                Symble.append(left_part)
        else:
            cur_state = State[-1]
            state_transform = LR1.get(cur_state, None)
            GOTO = state_transform.get(Symble[-1], None)
            if GOTO == None:
                output = 'Error'
                self.addTableRow(State, Symble, input_queue, output)
                break
            else:
                State.append(GOTO)
```

3 测试报告

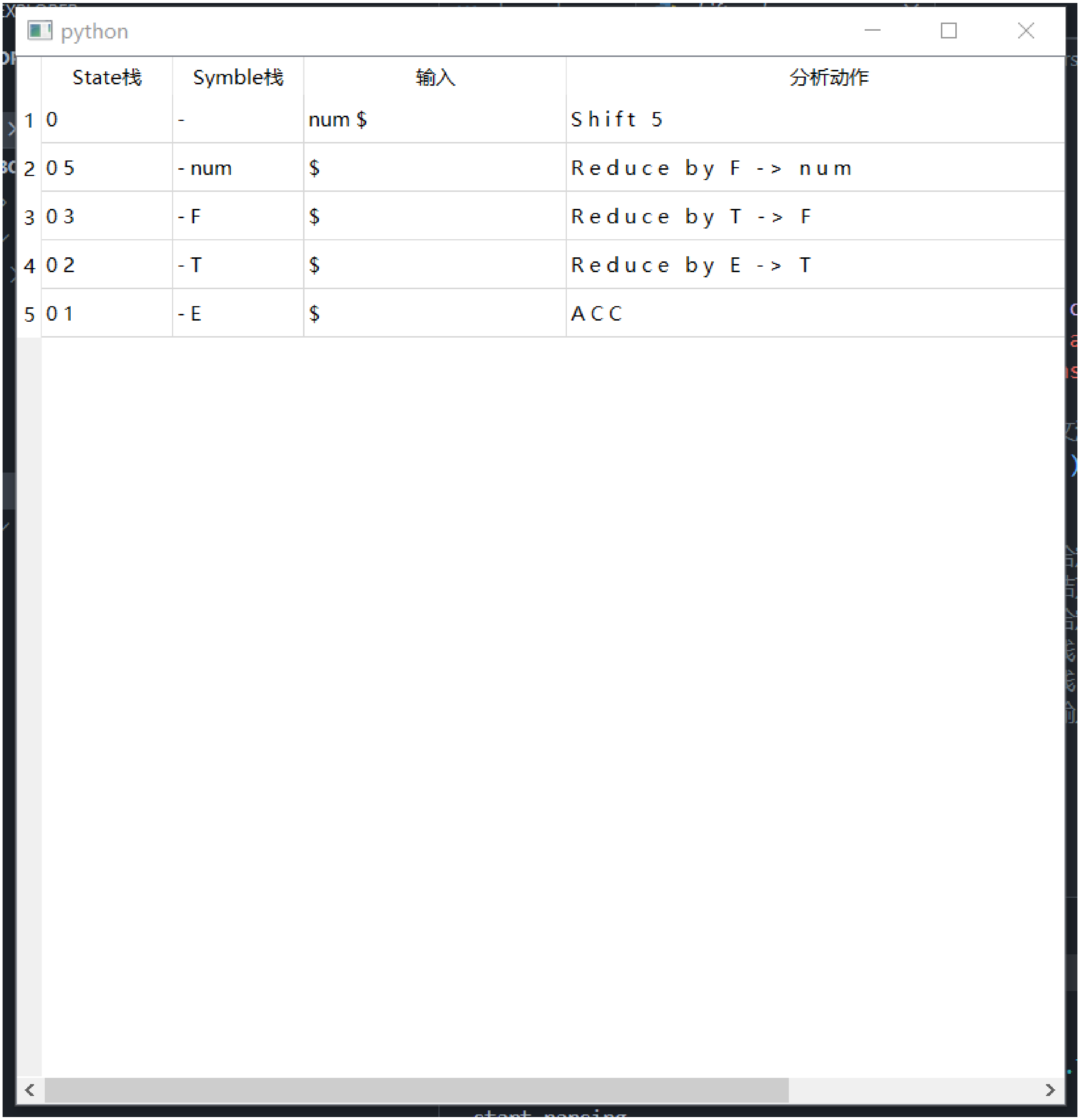
其中测试#1、测试#2、测试#3、测试#4为能被该文法描述的表达式
测试#5、测试#6、测试#7为不能被该文法描述的表达式

3.1 测试#1

输入:

```
num
```

输出:



	State栈	Symble栈	输入	分析动作
1	0	-	num $	Shift 5
2	0 5	- num	$	Reduce by F -> num
3	0 3	- F	$	Reduce by T -> F
4	0 2	- T	$	Reduce by E -> T
5	0 1	- E	$	ACC

3.2 测试#2

输入：

```
num + num - num
```

输出：

	State栈	Symble栈	输入	分析动作
1	0	-	num + num - num $	Shift 5
2	0 5	- num	+ num - num $	Reduce by F -> num
3	0 3	- F	+ num - num $	Reduce by T -> F
4	0 2	- T	+ num - num $	Reduce by E -> T
5	0 1	- E	+ num - num $	Shift 6
6	0 1 6	- E +	num - num $	Shift 5
7	0 1 6 5	- E + num	- num $	Reduce by F -> num
8	0 1 6 3	- E + F	- num $	Reduce by T -> F
9	0 1 6 15	- E + T	- num $	Reduce by E -> E + T
10	0 1	- E	- num $	Shift 7
11	0 1 7	- E -	num $	Shift 5
12	0 1 7 5	- E - num	$	Reduce by F -> num
13	0 1 7 3	- E - F	$	Reduce by T -> F
14	0 1 7 16	- E - T	$	Reduce by E -> E - T
15	0 1	- E	$	ACC

3.3 测试#3

输入:

```
( ( num + num ) / num - num ) * num
```

输出:

python

	State栈	Symble栈	输入	分析动作
1	0	-	((num + num) / num - num) * num $	Shift 4
2	0 4	- (	(num + num) / num - num) * num $	Shift 13
3	0 4 13	- ((	num + num) / num - num) * num $	Shift 14
4	0 4 13 14	- ((num	+ num) / num - num) * num $	Reduce by F -> num
5	0 4 13 12	- ((F	+ num) / num - num) * num $	Reduce by T -> F
6	0 4 13 11	- ((T	+ num) / num - num) * num $	Reduce by E -> T
7	0 4 13 24	- ((E	+ num) / num - num) * num $	Shift 20
8	0 4 13 24 20	- ((E +	num) / num - num) * num $	Shift 14
9	0 4 13 24 20 14	- ((E + num	) / num - num) * num $	Reduce by F -> num
10	0 4 13 24 20 12	- ((E + F	) / num - num) * num $	Reduce by T -> F
11	0 4 13 24 20 25	- ((E + T	) / num - num) * num $	Reduce by E -> E + T
12	0 4 13 24	- ((E	) / num - num) * num $	Shift 29
13	0 4 13 24 29	- ((E)	/ num - num) * num $	Reduce by F -> (E)
14	0 4 12	- (F	/ num - num) * num $	Reduce by T -> F
15	0 4 11	- (T	/ num - num) * num $	Shift 23
16	0 4 11 23	- (T /	num - num) * num $	Shift 14
17	0 4 11 23 14	- (T / num	- num) * num $	Reduce by F -> num
18	0 4 11 23 28	- (T / F	- num) * num $	Reduce by T -> T / F
19	0 4 11	- (T	- num) * num $	Reduce by E -> T
20	0 4 10	- (E	- num) * num $	Shift 21

	State栈	Symble栈	输入	分析动作
14	0 4 12	- (F	/ num - num) * num $	Reduce by T -> F
15	0 4 11	- (T	/ num - num) * num $	Shift 23
16	0 4 11 23	- (T /	num - num) * num $	Shift 14
17	0 4 11 23 14	- (T / num	- num) * num $	Reduce by F -> num
18	0 4 11 23 28	- (T / F	- num) * num $	Reduce by T -> T / F
19	0 4 11	- (T	- num) * num $	Reduce by E -> T
20	0 4 10	- (E	- num) * num $	Shift 21
21	0 4 10 21	- (E -	num) * num $	Shift 14
22	0 4 10 21 14	- (E - num	) * num $	Reduce by F -> num
23	0 4 10 21 12	- (E - F	) * num $	Reduce by T -> F
24	0 4 10 21 26	- (E - T	) * num $	Reduce by E -> E - T
25	0 4 10	- (E	) * num $	Shift 19
26	0 4 10 19	- (E)	* num $	Reduce by F -> (E)
27	0 3	- F	* num $	Reduce by T -> F
28	0 2	- T	* num $	Shift 8
29	0 2 8	- T *	num $	Shift 5
30	0 2 8 5	- T * num	$	Reduce by F -> num
31	0 2 8 17	- T * F	$	Reduce by T -> T * F
32	0 2	- T	$	Reduce by E -> T
33	0 1	- E	$	ACC

3.4 测试#4

输入:

```
( ( num - num + num ) * ( num / num ) )
```

输出:

python

	State栈	Symble栈	输入	分析
1	0	-	((num - num + num) * (num / num)) $	Shift 4
2	0 4	- (	(num - num + num) * (num / num)) $	Shift 13
3	0 4 13	- ((	num - num + num) * (num / num)) $	Shift 14
4	0 4 13 14	- ((num	- num + num) * (num / num)) $	Reduce by F -> num
5	0 4 13 12	- ((F	- num + num) * (num / num)) $	Reduce by T -> F
6	0 4 13 11	- ((T	- num + num) * (num / num)) $	Reduce by E -> T
7	0 4 13 24	- ((E	- num + num) * (num / num)) $	Shift 21
8	0 4 13 24 21	- ((E -	num + num) * (num / num)) $	Shift 14
9	0 4 13 24 21 14	- ((E - num	+ num) * (num / num)) $	Reduce by F -> num
10	0 4 13 24 21 12	- ((E - F	+ num) * (num / num)) $	Reduce by T -> F
11	0 4 13 24 21 26	- ((E - T	+ num) * (num / num)) $	Reduce by E -> E - T
12	0 4 13 24	- ((E	+ num) * (num / num)) $	Shift 20
13	0 4 13 24 20	- ((E +	num) * (num / num)) $	Shift 14
14	0 4 13 24 20 14	- ((E + num	) * (num / num)) $	Reduce by F -> num
15	0 4 13 24 20 12	- ((E + F	) * (num / num)) $	Reduce by T -> F
16	0 4 13 24 20 25	- ((E + T	) * (num / num)) $	Reduce by E -> E + T
17	0 4 13 24	- ((E	) * (num / num)) $	Shift 29
18	0 4 13 24 29	- ((E)	* (num / num)) $	Reduce by F -> (E)
19	0 4 12	- (F	* (num / num)) $	Reduce by T -> F
20	0 4 11	- (T	* (num / num)) $	Shift 22

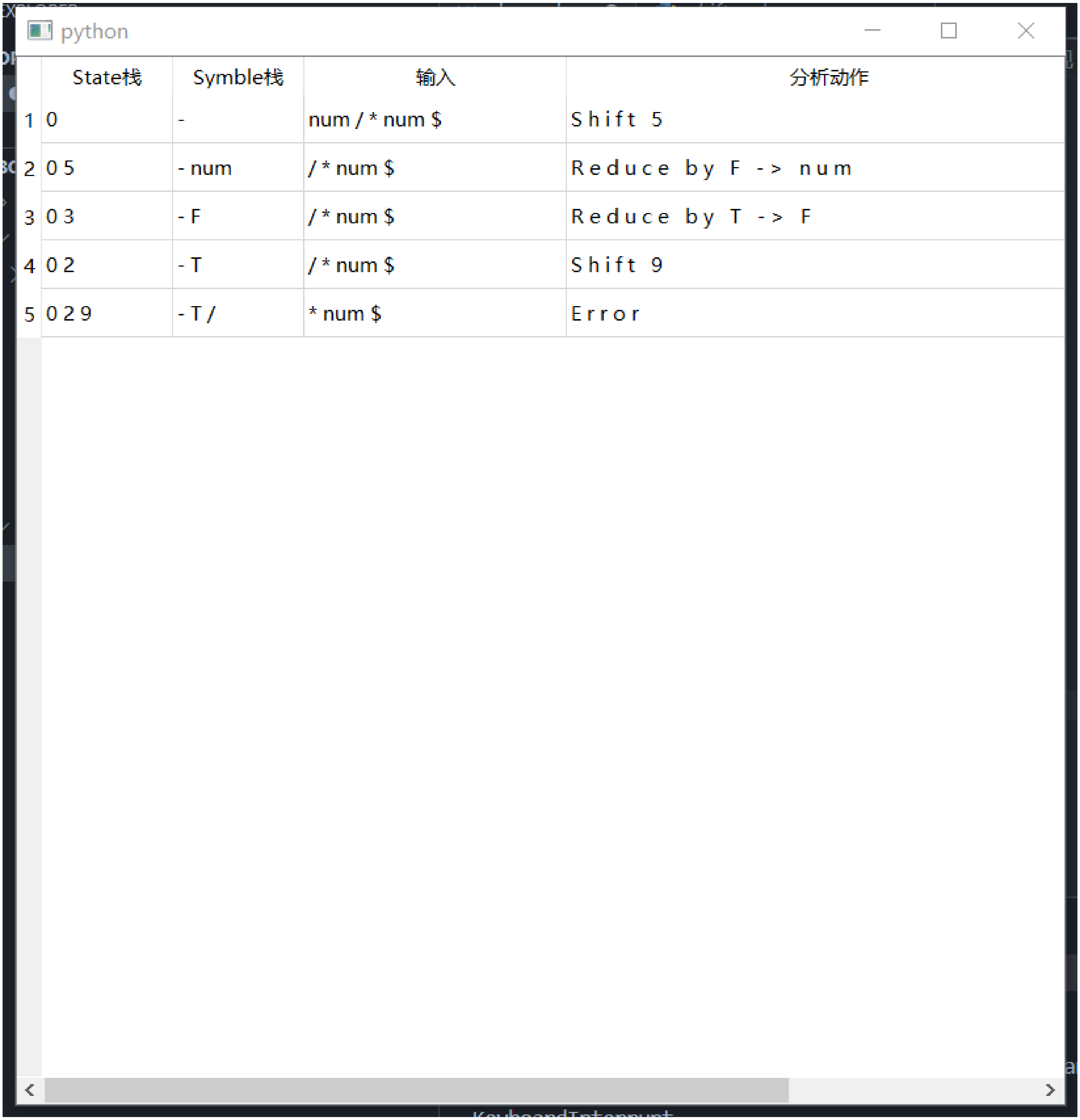
	State栈	Symble栈	输入	分析
19	0 4 12	- (F	* (num / num)) $	Reduce by T -> F
20	0 4 11	- (T	* (num / num)) $	Shift 22
21	0 4 11 22	- (T *	(num / num)) $	Shift 13
22	0 4 11 22 13	- (T * (	num / num)) $	Shift 14
23	0 4 11 22 13 14	- (T * (num	/ num)) $	Reduce by F -> num
24	0 4 11 22 13 12	- (T * (F	/ num)) $	Reduce by T -> F
25	0 4 11 22 13 11	- (T * (T	/ num)) $	Shift 23
26	0 4 11 22 13 11 23	- (T * (T /	num)) $	Shift 14
27	0 4 11 22 13 11 23 14	- (T * (T / num	)) $	Reduce by F -> num
28	0 4 11 22 13 11 23 28	- (T * (T / F	)) $	Reduce by T -> T / F
29	0 4 11 22 13 11	- (T * (T	)) $	Reduce by E -> T
30	0 4 11 22 13 24	- (T * (E	)) $	Shift 29
31	0 4 11 22 13 24 29	- (T * (E)	) $	Reduce by F -> (E)
32	0 4 11 22 27	- (T * F	) $	Reduce by T -> T * F
33	0 4 11	- (T	) $	Reduce by E -> T
34	0 4 10	- (E	) $	Shift 19
35	0 4 10 19	- (E)	$	Reduce by F -> (E)
36	0 3	- F	$	Reduce by T -> F
37	0 2	- T	$	Reduce by E -> T
38	0 1	- E	$	ACC

3.5 测试#5

输入：

```
num / * num
```

输出：



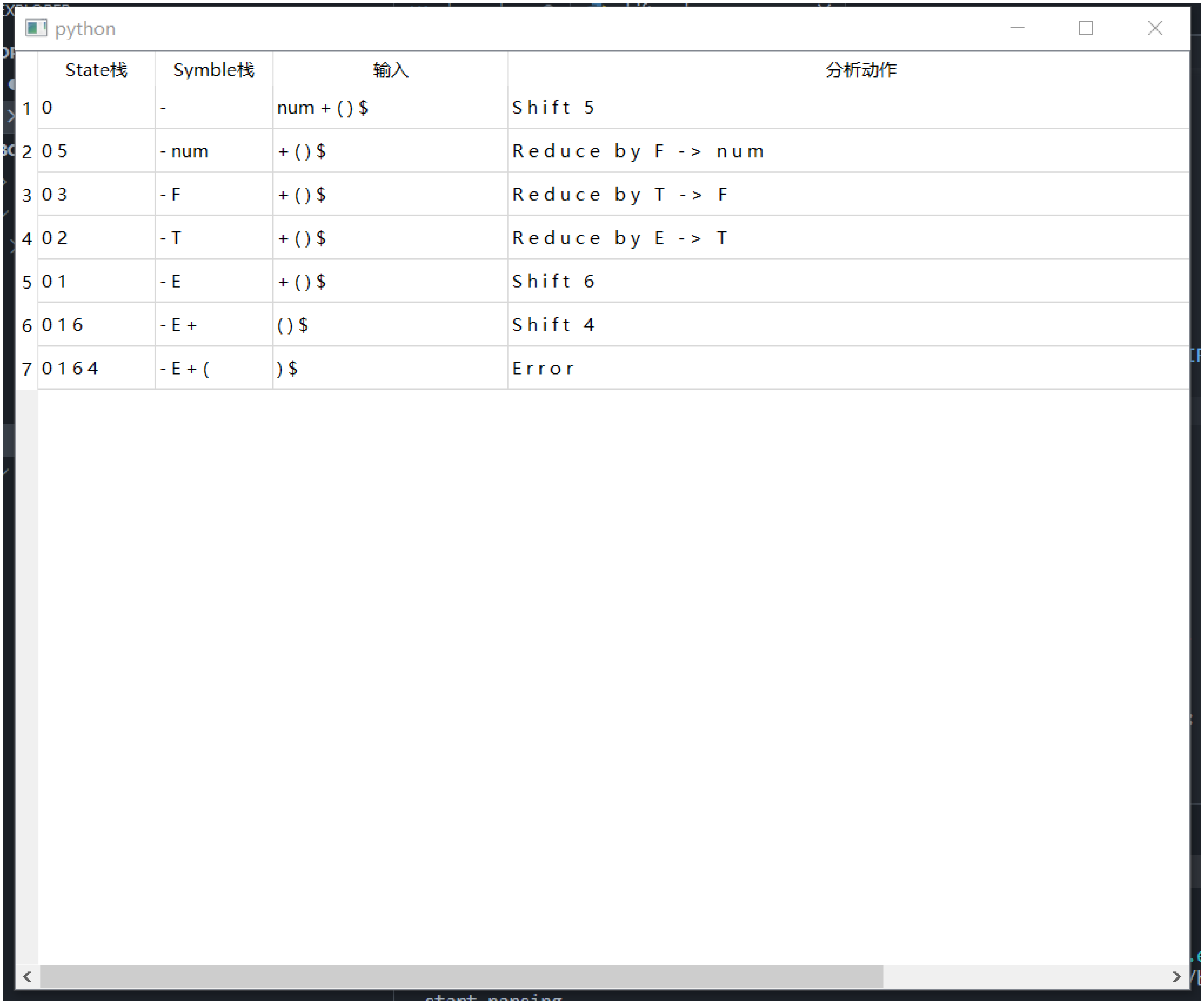
	State栈	Symble栈	输入	分析动作
1	0	-	num / * num $	Shift 5
2	0 5	- num	/ * num $	Reduce by F -> num
3	0 3	- F	/ * num $	Reduce by T -> F
4	0 2	- T	/ * num $	Shift 9
5	0 2 9	- T /	* num $	Error

3.6 测试#6

输入：

```
num + ( )
```

输出：



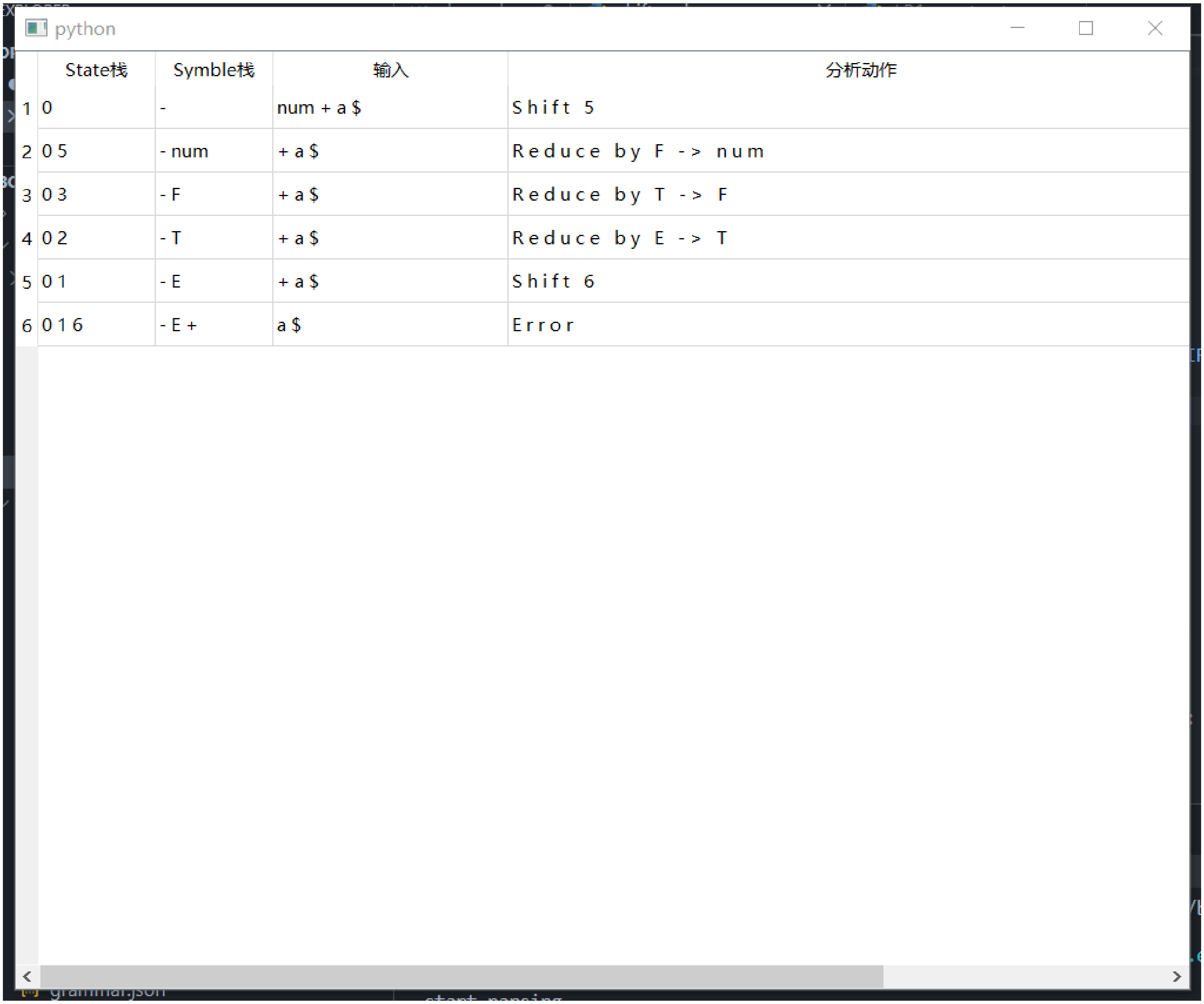
	State栈	Symble栈	输入	分析动作
1	0	-	num + ( ) $	Shift 5
2	0 5	- num	+ ( ) $	Reduce by F -> num
3	0 3	- F	+ ( ) $	Reduce by T -> F
4	0 2	- T	+ ( ) $	Reduce by E -> T
5	0 1	- E	+ ( ) $	Shift 6
6	0 1 6	- E +	( ) $	Shift 4
7	0 1 6 4	- E + (	) $	Error

3.7 测试#7

输入:

```
num + a
```

输出:



	State栈	Symble栈	输入	分析动作
1	0	-	num + a $	Shift 5
2	0 5	- num	+ a $	Reduce by F -> num
3	0 3	- F	+ a $	Reduce by T -> F
4	0 2	- T	+ a $	Reduce by E -> T
5	0 1	- E	+ a $	Shift 6
6	0 1 6	- E +	a $	Error

4 实验总结与心得

- 本次实验中我编写了一个 $LR(1)$ 语法分析程序，使我对自底向上语法分析的流程更加清楚，对相关知识点的掌握更加牢固。
- 此程序的架构主要采用了模块化编程，各个模块分开测试，模块功能划分比较明确，架构并不复杂。
- 主要难点在数据结构的设计和算法的实现方面，尤其是对于 $LR(1)$ 项目和项目集，表示方式需要既准确又要便于计算，实现的难度较大。多次考量之后选择了 $python$ 的 $list$ 数据结构
- 此外，我的语法分析程序仍然存在许多待改进的地方，比如在构造 $LR(1)$ 分析表时对可能存在的错误处理不够全面，没有验证对于存在空产生式的CFG能否正确地进行 $LR(1)$ 分析，由于时间所限，这些情况我无法一一考虑周全。
- 在完成实验内容后我根据$PyQt5$框架实现了简单地图形化界面，让语法分析过程更加直观。这也是我选择$python$完成此次实验的初衷之一(语法分析实现过程的核心算法部分全部是我亲自手写的代码，没有借助$python$中任何工具)。

- 本次实验用模拟算法模拟自己实现 $LR(1)$ 语法分析的过程，除了让我对课内知识有了更多的认识，也使我的 $python$ 编程能力得到提高，我从中收获颇丰。

5 源代码

5.1 $extended_grammar_constructor.py$文件如下：

```python
# 该模块用于将输入的文法改写为拓广文法

import json

# 返回改写后拓广文法的字典
def constructor(grammar_file: str) -> dict:
    # 读入输入的文法
    with open(grammar_file, 'r') as json_file:
        grammar = json.load(json_file)

    # 起始符改为 E'
    start_symbol = 'E\''

    # 将新的起始符 E' 加入非终结符集合中
    non_terminal = grammar.get('non_terminal', None)
    non_terminal.insert(0, 'E\'')

    # 加入一条产生式 E' -> E
    production = grammar.get('production', None)
    production.insert(0, 'E\' -> E')

    grammar.update({'start_symbol' : start_symbol,
                    'non_terminal' : non_terminal,
                    'production':production})
    return grammar

if __name__ == "__main__":
    grammar = constructor('grammar.json')
    # 把拓广文法字典写入文件extended_grammar.json
    with open("extended_grammar.json", 'w') as json_file:
        json.dump(grammar, json_file, indent=4)
```

5.2 $FIRST_calculator.py$文件如下：

```python
# FIRST集合计算模块
'''
API:
constructor(grammar_file: str) -> dict:
    输入：需构造FIRST集的文法所在的文件
    输出：构造出的FIRST集合的字典：key为文法符号，value为FIRST集合元素构成的列表

get_string_FIRST(string: list, FIRST: dict) -> list:
    输入：要求的文法符号串形成的列表，求FIRST集时已知的各个文法符号的FIRST集
    输出：求得的FIRST集元素构成的列表
'''

import json
from typing import Tuple
from extended_grammar_constructor import constructor as EGC

# 将给定产生式的左部和右部分开
def get_symbol(production: str) -> Tuple[str, list]:
    left_part, right_part = production.split('->')
    left_part = left_part.strip()
    right_symbols = right_part.strip().split(' ')
    return left_part, right_symbols

# 根据给定的FIRST集计算当前string的FIRST集
def get_string_FIRST(string: list, FIRST: dict) -> list:
    res = []
    empty_stringable = True
    for symbol in string:
        if 'ε' not in FIRST.get(symbol):
            res.extend(FIRST.get(symbol))
            empty_stringable = False
            break
        else:
            # 加入除 ε 外所有元素
            res.extend(x for x in FIRST.get(symbol) if x != 'ε')
    # 所有文法符号均可为空串，则FIRST中包含ε
    if empty_stringable:
        res.append('ε')
    # 去重
    return list(set(res))
```

```python
# 计算给定文法各个符号的FIRST
def constructor(grammar: dict) -> dict:
    # 文法符号的FIRST集的字典，映射关系为文法符号->FIRST集合
    FIRST = {}

    # 对终结符 a 有 FIRST(a) = {a}
    terminal = grammar.get('terminal', None)
    for symbol in terminal:
        FIRST[symbol] = [symbol]

    # 构造非终结符的FIRST集
    non_terminal = grammar.get('non_terminal', None)
    for symbol in non_terminal:
        FIRST[symbol] = []
    production = grammar.get('production', None)
    while True:
        modified = False
        for p in production:
            left, right = get_symbol(p)
            # print(left, right)
            if right == ['ε']: # 产生式形如 A -> ε
                if 'ε' not in FIRST.get(left):
                    FIRST.get(left).append('ε')
                    modified = True
            else:
                added_symbols = get_string_FIRST(right, FIRST)
                for symbol in added_symbols:
                    if symbol not in FIRST.get(left):
                        FIRST.get(left).append(symbol)
                        modified = True
        if modified == False:
            break
    return FIRST

if __name__ == "__main__":
    grammar = EGC('grammar.json')
    FIRST = constructor(grammar)
    # 把FIRST集写入文件FIRST.json
    with open("FIRST.json", 'w') as json_file:
        json.dump(FIRST, json_file, indent=4)
```

5.3 $LR1_constructor$文件如下

```python
# 构造LR1分析表

import json
from typing import Tuple
from collections import Counter
from extended_grammar_constructor import constructor as extended_grammar_constructor
from FIRST_calculator import constructor as FIRST_constructor
from FIRST_calculator import get_string_FIRST


# 将给定产生式(字符串表示)拆成左部和右部
def get_symbol(production: str) -> Tuple[str, list]:
    left_part, right_part = production.split('->')
    left_part = left_part.strip()
    right_symbols = right_part.strip().split(' ')
    return left_part, right_symbols


# 将给定的产生式集合拆成左部集合(left = ['X', 'Y'])和右部集合(right = [[], []])
def split_production(production: list) -> Tuple[list, list]:
    left = []
    right = []
    for p in production:
        left_part, right_part = get_symbol(p)
        left.append(left_part)
        right.append(right_part)
    return left, right

# 计算项目集闭包
def closure(item: list, grammar: dict, FIRST: dict) -> list:
    '''
    item = [[production, dot, lookahead]]
    其中 dot 表示点在产生式右部第几个文法符号的前面(从0开始编号),
    特别的, 当 dot = len(右部) 时表示规约项目
    '''
    grammar_production = grammar.get('production')
    grammar_left = grammar.get('left')
    grammar_right = grammar.get('right')
    grammar_non_terminal = grammar.get('non_terminal')
    count = 0
    while count < len(item):
```

```python
        i = item[count]
        cur_production_index = grammar_production.index(i[0]) # 当前要处理的产生式的索引
        if i[1] == len(grammar_right[cur_production_index]): # 是一个归约项目
            pass
        elif grammar_right[cur_production_index][i[1]] in grammar_non_terminal: # 是一个待约项目
            # 对于形如 A -> α · B, 找到所有 B 为左部的产生式的 index
            indices = []
            for index, value in enumerate(grammar_left):
                if value == grammar_right[cur_production_index][i[1]]:
                    indices.append(index)
            # 计算向前看符号串
            r = grammar_right[cur_production_index][(i[1] + 1):]
            r.append(i[2])
            lookahead = get_string_FIRST(r, FIRST)
            new_item = []
            for index in indices:
                for symbol in lookahead:
                    new_item.append([grammar_production[index], 0, symbol])
            for it in new_item:
                if it not in item:
                    item.append(it)
        else: # 是一个移进项目
            pass
        count += 1
    return item

# 给定项目 item 找出项目集中所有能在相同输入进行转移的项目编号
def get_same_items_index(item: list, cur_item: list, production: list, right: list) -> list:
    indices = []
    production_index = production.index(item[0])
    symbol = right[production_index][item[1]]  # 下一个输入符号
    for index, item in enumerate(cur_item):
        if len(right[production.index(item[0])]) <= item[1]:
            continue
        elif right[production.index(item[0])][item[1]] == symbol:
            indices.append(index)
    return indices

# 计算new_item是否在项目集族items中位置，若小于len(items)则已存在，返回在items中索引
def get_item_index(new_item: list, items: list) -> int:
    new_item_hashable = frozenset(tuple(it) for it in new_item)
    for index, existing_item in enumerate(items):
        existing_item_hashable = frozenset(tuple(it) for it in existing_item)
```

```python
        if existing_item_hashable == new_item_hashable:
            return index
    return len(items)

# 构造该文法的LR(1)项目集规范族
def constructor(grammar: dict, FIRST: dict) -> dict:
    # 将产生式拆成左部和右部，方便后续操作
    production = grammar.get('production', None)
    left, right = split_production(production)
    grammar['left'] = left
    grammar['right'] = right
    terminal = grammar.get('terminal', None)
    LR1 = {} # LR(1)分析表
    '''
    LR1 = {state1 : {'A' : state2, 'a' : ['R'/'S', index]}}
    表示在state1遇到非终结符转移到state2,
    遇到终结符a根据第index条产生式规约或
    移入并转移到状态index
    '''
    items = [] # 项目集规范族
    equal_item = {}
    items.append(closure([["E' -> E", 0, '$']], grammar, FIRST))
    state_index = 0
    while state_index < len(items):
        cur_item = items[state_index]
        # 计算状态state_index的goto和action
        LR1[state_index] = {}
        is_checked = [False] * len(cur_item)
        for i, item in enumerate(cur_item):
            if is_checked[i]: # 已经加入新的项目了
                continue
            else:
                production_index = production.index(item[0])
                condition_1 = item[1] == len(right[production_index])
                condition_2 = right[production_index] == ['ε']
                if condition_1 or condition_2: # 是一个归约项目
                    if LR1[state_index].get(item[2], None) == None:
                        if production_index == 0:
                            LR1[state_index][item[2]] = 'ACC'
                        else:
                            LR1[state_index][item[2]] = ['R', production_index]
                    elif LR1[state_index][item[2]] != ['R', production_index]:
                        print('归约存在冲突')
```

```
                return None # 存在冲突，不是LR(1)文法，返回None
            elif right[production_index][item[1]] in terminal: # 是一个移进项目
                # 对于 a ， 找到项目集中所有形如A -> α · a β的项目，加入项目集规范族
                indices = get_same_items_index(item, cur_item, production, right)
                new_item = []
                for index in indices:
                    is_checked[index] = True
                    new_item.append([cur_item[index][0],
                                     cur_item[index][1] + 1,
                                     cur_item[index][2]])
                new_item = closure(new_item, grammar, FIRST)
                new_item_index = get_item_index(new_item, items)
                symbol = right[production_index][item[1]]
                if LR1[state_index].get(symbol, None) == None:
                    LR1[state_index][symbol] = ['S', new_item_index]
                elif LR1[state_index][symbol] != ['S', new_item_index]:
                    print('移进项目存在冲突')
                    return None
                if new_item_index == len(items):
                    items.append(new_item)
            else: # 是一个待约项目
                # 对于 B ，找到项目集中所有形如A -> α · B β的项目，加入项目集规范族
                indices = get_same_items_index(item, cur_item, production, right)
                new_item = []
                for index in indices:
                    is_checked[index] = True
                    new_item.append([cur_item[index][0],
                                     cur_item[index][1] + 1,
                                     cur_item[index][2]])
                new_item = closure(new_item, grammar, FIRST)
                new_item_index = get_item_index(new_item, items)
                symbol = right[production_index][item[1]]
                if LR1[state_index].get(symbol, None) == None:
                    LR1[state_index][symbol] = new_item_index
                elif LR1[state_index][symbol] != new_item_index:
                    print('待约项目存在冲突')
                    return None
                if new_item_index == len(items):
                    items.append(new_item)
        state_index += 1
    with open("closure.json", 'w') as json_file:
        json.dump(items, json_file, indent=2)
    return LR1
```

```python
if __name__ == "__main__":
    with open('extended_grammar.json', 'r') as json_file:
        grammar = json.load(json_file)
    FIRST = FIRST_constructor(grammar)
    FIRST['$'] = ['$'] # 加入结束符 $

    LR1 = constructor(grammar, FIRST)

    # 把LR(1)分析表写入文件LR1.json
    with open("LR1.json", 'w') as json_file:
        json.dump(LR1, json_file, indent=2)
```

5.4 $shift_reduce_parser.py$文件如下

```
# LR(1) 移入-归约分析

import sys
from PyQt5.QtWidgets import QApplication, QMainWindow, QTableWidget, QTableWidgetItem
from PyQt5.QtCore import Qt
from collections import deque

from extended_grammar_constructor import constructor as EGC
from FIRST_calculator import constructor as FC
from LR1_constructor import constructor as LR1C

grammar = EGC('grammar.json')      # 给定文法的拓广文法
non_terminal = grammar.get('non_terminal')
terminal = grammar.get('terminal')
production = grammar.get('production')
FIRST = FC(grammar)                # 构造给定文法所有文法符号的FIRST集
FIRST['$'] = ['$']                 # 加入结束符 $
LR1 = LR1C(grammar, FIRST)         # 构造给定文法的LR(1)分析表
State = [0]                        # 状态栈
Symble = ['-']                     # 符号栈
user_input = input()               # 用户输入
user_input = user_input.split()
input_queue = deque()
for word in user_input:
    input_queue.append(word)
input_queue.append('$')
output = None

class LR1Parsing(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()
    def initUI(self):
        global grammar, non_terminal, terminal, production, FIRST, \
        LR1, State, Symble, input_queue, output
        self.setGeometry(100, 100, 1000, 800)
        self.tableWidget = QTableWidget(self)
        self.tableWidget.setGeometry(0, 0, 1000, 800)
        self.tableWidget.setColumnCount(4)
        self.tableWidget.setColumnWidth(0, 100)
        self.tableWidget.setColumnWidth(1, 100)
```

```
        self.tableWidget.setColumnWidth(2, 200)
        self.tableWidget.setColumnWidth(3, 600)
        self.tableWidget.setHorizontalHeaderLabels(['State栈', 'Symble栈', '输入', '分析动作'])
        print('start parsing')
        self.startParsing()
    def addTableRow(self, State, Symble, input_queue, output):
        rowPosition = self.tableWidget.rowCount()
        self.tableWidget.insertRow(rowPosition)
        state = ' '.join(map(str, State))
        symble = ' '.join(Symble)
        cur_input = ' '.join(input_queue)
        cur_output = ' '.join(output)
        self.tableWidget.setItem(rowPosition, 0, QTableWidgetItem(state))
        self.tableWidget.setItem(rowPosition, 1, QTableWidgetItem(symble))
        self.tableWidget.setItem(rowPosition, 2, QTableWidgetItem(cur_input))
        self.tableWidget.setItem(rowPosition, 3, QTableWidgetItem(cur_output))
    def startParsing(self):
        while True:
            if len(State) == len(Symble):
                cur_state = State[-1]
                state_transform = LR1.get(cur_state, None)
                input_stack_top = input_queue[0]
                ACTION = state_transform.get(input_stack_top, None)
                if ACTION == None:
                    output = 'Error'
                    self.addTableRow(State, Symble, input_queue, output)
                    break
                elif ACTION == 'ACC':
                    output = 'ACC'
                    self.addTableRow(State, Symble, input_queue, output)
                    break
                elif ACTION[0] == 'S':
                    output = 'Shift ' + str(ACTION[1])
                    self.addTableRow(State, Symble, input_queue, output)
                    input_queue.popleft()
                    State.append(ACTION[1])
                    Symble.append(input_stack_top)
                else:
                    output = 'Reduce by ' + production[ACTION[1]]
                    self.addTableRow(State, Symble, input_queue, output)
                    left_part, right_part = production[ACTION[1]].split('->')
                    left_part = left_part.strip()
                    right_symbols = right_part.strip().split(' ')
```

```
                for symbol in reversed(right_symbols):
                    if symbol == Symble[-1]:
                        State.pop()
                        Symble.pop()
                    else:
                        output = 'Error'
                        self.addTableRow(State, Symble, input_queue, output)
                        break
                Symble.append(left_part)
        else:
            cur_state = State[-1]
            state_transform = LR1.get(cur_state, None)
            GOTO = state_transform.get(Symble[-1], None)
            if GOTO == None:
                output = 'Error'
                self.addTableRow(State, Symble, input_queue, output)
                break
            else:
                State.append(GOTO)

if __name__ == '__main__':
    if LR1 == None:
        print('该文法不是LR(1)文法，无法进行LR(1)分析')
    else:
        app = QApplication(sys.argv)
        lr1_parsing = LR1Parsing()
        lr1_parsing.show()
        sys.exit(app.exec_())
```

以下是调试过程中写的代码

5.5 $grammar_writer.py$文件如下

```python
# 该模块用于输入待分析的文法

import json

# 文法字典
grammar = {}

terminal = ['+', '-', '*', '/', '(', ')', 'num']    # 终结符
non_terminal = ['E', 'T', 'F']                      # 非终结符
start_symbol = 'E'                                  # 起始符
production = []                                     # 产生式

# 加入产生式
production.append('E -> E + T')
production.append('E -> E - T')
production.append('E -> T')
production.append('T -> T * F')
production.append('T -> T / F')
production.append('T -> F')
production.append('F -> ( E )')
production.append('F -> num')

# 将文法写入文法字典
grammar['terminal'] = terminal
grammar['non_terminal'] = non_terminal
grammar['start_symbol'] = start_symbol
grammar['production'] = production

# 把文法字典写入文件grammar.json
with open("grammar.json", 'w') as json_file:
    json.dump(grammar, json_file, indent=4)
```

5.6 $item_debugger.py$文件如下

```python
# 该文件用于验证LR1_construtor是否正确构造了项目集规范族
import json

items = []

with open("closure.json", 'r') as json_file:
    items = json.load(json_file)

index = 0

for item in items:
    print(f'I{index}:')
    index = index + 1
    a = []
    b = []
    for it in item:
        production = it[0]
        production = production + ' '
        dot = it[1] + 1
        lookahead = it[2]
        arrow_index = production.find('->')
        spaces_after_arrow = [pos for pos, char in enumerate\
                              (production[arrow_index + 2:]) if char.isspace()]
        if len(spaces_after_arrow) >= dot:
            target_space_index = arrow_index + 2 + spaces_after_arrow[dot - 1]
        modified_str = production[:target_space_index] + '·' \
        + production[target_space_index:]
        if modified_str not in a:
            a.append(modified_str)
            b.append([lookahead])
        else:
            num = a.index(modified_str)
            b[num].append(lookahead)
    cnt = 0
    for aa in a:
        print(f'{aa},  {b[cnt]}')
        cnt = cnt + 1
```