LL(1) 语法分析程序的设计与实现

杨晨
学号 2021212171
北京邮电大学计算机学院
日期：2023 年 11 月 15 日

1 概述

1.1 实验内容

编写语法分析程序，实现对算术表达式的语法分析。要求所分析算术表达式由如下的文法产生。

$$E \to E+T|E-T|T$$
$$T \to T*F|T/F|F$$
$$F \to (E)|num$$

要求在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

编写 LL(1) 语法分析程序，实验要求和实现方法要求如下：

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。

1.2 开发环境

- Windows10
- PyCharm 2023.2.4 (Professional Edition)

2 程序的功能模块划分

2.1 文法 Grammar 类

2.1.1 定义

首先，考虑文法的特点，它的终结符和非终结符号，字母符号之间都具有互异、唯一的特点，所以可以用集合来存储。其次，考虑产生式，一个非终结符号，可能会有多个右部产生式，同时，为了方便查找产生式，可以用字典来存储产生式，字典的键值是左部符号，对应的值是一个列表，包含这个符号的所有产生式。

```
class Grammar:
    def __init__(self):
        self.non_terminals = set()
        self.terminals = set()
        self.productions = {}
        self.start_symbol = None
```

Grammar 类的构造函数 __init__初始化了文法对象的各个属性：

- non_terminals 是一个集合，用于存储非终结符（non-terminal）。
- terminals 是一个集合，用于存储终结符（terminal）。
- productions 是一个字典，用于存储产生式（production）。字典的键是非终结符，值是该非终结符对应的产生式列表，其中列表中每个元素是一个字符串，对应一个产生式。
- start_symbol 是一个变量，用于文法存储起始符号。

2.1.2 添加产生式

由于我们通常约定，终结符号是大写字母，而非终结符合是小写字母，所以可以利用这个特点来填充终结符号集合和非终结符号集合

而产生式的右部长度不能确定，为了简单起见，采用一次插入一个产生式的方法，即如果产生式是形如

$$E \to E+T|E-T|T$$

这样的形式，那么则需要调用 3 次添加函数

$$add\_production('E','E+T')$$

$$add\_production('E','E-T')$$

$$add\_production('E','T')$$

```
# 添加产生式，non_terminal是非终结符，production是产生式
def add_production(self, non_terminal, production):
    if non_terminal not in self.productions:
        self.productions[non_terminal] = []

    self.productions[non_terminal].append(production)
    self.non_terminals.add(non_terminal)

    if production == "num":
        self.terminals.add("num")
    else:
        for symbol in production:
            if symbol.isalpha() and not symbol.islower():  # 是大写字母
                self.non_terminals.add(symbol)
            else:
```

```
                self.terminals.add(symbol)
```

add_production方法用于向文法中添加产生式。它接受两个参数：non_terminal（非终结符）和 production（产生式）。该方法将产生式添加到相应的非终结符的产生式列表中，并更新 non_terminals 和 terminals 集合。

2.1.3 消除左递归

左递归是指产生式右部以相同的非终结符开头的情况。这会对后续的 LL1 分析造成无限递归的死循环，所以，我们需要对文法做消除左递归的操作。消除左递归的方法如下

对于有如下产生式的非终结符 A：

$$A \to A\alpha_1|A\alpha_2|\dots|A\alpha_n|\beta_1|\beta_2|\dots|\beta_m$$

其中，$\beta_i(i=1,2,\dots,m)$ 不以 A 打头。用如下产生式替代：

$$A \to \beta_1 A'|\beta_2 A'|\dots|\beta_m A'$$
$$A' \to \alpha_1 A'|\alpha_2 A'|\dots|\alpha_n A'|\varepsilon$$

```
# 消除左递归
def eliminate_left_recursion(self):
    non_terminals = list(self.non_terminals)

    for left_element in non_terminals:
        remove_list = []
        for right_list in self.productions[left_element]:
            if right_list[0] == left_element:
                remove_list.append(right_list)
                if left_element + "'" not in self.non_terminals:
                    self.non_terminals.add(left_element + "'")
                    self.productions[left_element + "'"] = []
        if remove_list:  # 如果remove_list不为空
            for right_str in remove_list:
                self.productions[left_element].remove(right_str)
                self.productions[left_element + "'"].append(
                    right_str[1:] + left_element + "'"
                )
            self.productions[left_element + "'"].append("ε")
            self.terminals.add("ε")
            self.productions[left_element] = [
                value + left_element + "'"
                for value in self.productions[left_element]
            ]
```

`eliminate_left_recursion`方法用于消除文法中的左递归。

该方法遍历文法中的每个非终结符，并检查其产生式列表中的产生式是否存在左递归。如果存在左递归，则会对产生式进行修改，将左递归的部分替换为新的非终结符。同时，新的非终结符会被添加到 `non_terminals` 集合中，并且会添加一个产生空串（ϵ）的产生式。最后，原始的产生式也会进行修改，以引用新的非终结符。

2.1.4 消除左公因子

在进行 LL(1) 分析时，如果文法中存在左公因子，会导致分析表中的某些表项具有两个或更多的元素，从而需要进行回溯操作。为了避免这种情况，我们可以通过提取左公因子的方法，将文法转换为不含左公因子的形式。这样，就可以得到满足 LL(1) 文法的要求，且不需要回溯操作的文法。消除方法如下：

对于每个非终结符 A，找出它的两个或更多候选式的最长公共前缀 α，如果 $\alpha \neq \varepsilon$，有如下产生式：

$$A \to \alpha\beta_1|\alpha\beta_2|\ldots|\alpha\beta_n|\gamma_1|\gamma_2|\ldots|\gamma_m$$

其中，$\gamma_i(i = 1, 2, \ldots, m)$ 表示不以 α 打头的表达式。用如下产生式替代：

$$A \to \alpha A'|\gamma_1|\gamma_2|\ldots|\gamma_m$$
$$A' \to \beta_1|\beta_2|\ldots|\beta_n$$

```
# 消除左公因子
def eliminate_left_factor(self):
    new_non_terminals = self.non_terminals  # 新的非终结符
    new_productions = {}  # 新的产生式
    while True:
        tmp_productions = {}  # 用于存储新的产生式
        for left_element in self.productions:  # 遍历产生式
            right_list = self.productions[left_element]  # 右侧产生式列表
            right_head = []  # 右侧产生式的首符号列表
            for right_str in right_list:
                right_head.append(right_str[0])
            for i in range(len(right_head)):  # 遍历右侧产生式的首符号
                for j in range(i + 1, len(right_head)):  # 遍历右侧产生式的
                    首符号
                    if (
                        right_head[i] in self.non_terminals
                        and right_head[j] in self.non_terminals
                    ):  # 都是非终结符
                        continue
                    if right_head[i] == right_head[j]:  # 首符号相同,消除左
                        公因子
```

```
                        new_non_terminals.add(left_element + "'")
                        if left_element + "'" not in tmp_productions:
                            tmp_productions[left_element + "'"] = set()
                        self.productions[left_element].remove(right_list[i
                            ])
                        self.productions[left_element].remove(right_list[j
                            ])
                        tmp_productions[left_element + "'"].add(right_list[
                            i][1:])
                        tmp_productions[left_element + "'"].add(right_list[
                            j][1:])
                        self.productions[left_element].add(
                            right_head[i] + left_element + "'"
                        )
        new_productions.update(self.productions)  # 更新产生式
        if tmp_productions == {}:  # 如果没有新的产生式，退出循环
            break
    self.productions = new_productions.copy()  # 更新产生式
```

2.1.5 文法格式化输出

为了便于观察和调试，我还设计了将文法按照标准格式进行输出的方法

```
# 输出文法
def print_grammar(self):
    for non_terminal in self.productions:
        print(non_terminal + " -> " + " | ".join(self.productions[
            non_terminal]))
    print()
```

可以在需要进行验证，或者调试的地方调用该方法，查看当前的文法

2.2 LL1 分析器

2.2.1 定义

考虑到 First 集和 Follow 集都和非终结符相对应，为了便于查找，所以用字典来存储。其中，由于每个非终结符合的 First 集里，元素都是互不相等且唯一的，所以，字典的键是非终结符，所对应的值是一个集合，集合中包含这个非终结符的 First 集的元素。同理，Follow 集的设计也是如此

对于预测分析表，由于它是一个二维表格，我们仍然可以用字典来存储，其中键是一个二元元组，包括非终结符和终结符，值是产生式右部。因为产生式的左部就是表中纵轴上的非终结符，故我们只需要记录产生式右部即可

```
class LL1Parser:
    def __init__(self):
        self.first = {}
        self.follow = {}
        self.predictions_table = {}
```

`LL1Parser` 类的构造函数 `__init__`初始化了文法对象的各个属性：

- `first` 是一个字典，用于存储每个非终结符的 First 集
- `follow` 是一个字典，用于存储每个非终结符的 Follow 集
- `predictions_table` 是一个字典，用于存储预测分析表

2.2.2 计算 First 集

构建非终结符和候选式的 FIRST 集，方法如下：

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

$$A \to Y_1 Y_2 \dots Y_k$$

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，如果没有求出它的 FIRST 集，则递归求解。之后，α 的 FIRST 集并上 Y_i 的 FIRST 集。此后检查 Y_i 的 FIRST 集中是否包含 ε（即是否能推导出 ε），若不包含，则终止遍历。

最后，A 的 FIRST 集为各个候选式的 FIRST 集的并。

```
# 计算first集
def compute_first(self, grammar):
    for non_terminal in grammar.non_terminals:  # 初始化
        self.first[non_terminal] = set()
    for terminal in grammar.terminals:
        self.first[terminal] = set()

    for non_terminal in grammar.non_terminals:
        self.calculate_first(grammar, non_terminal)

    for non_terminal in grammar.non_terminals:  # 输出first集
        print("First({}) = {}".format(non_terminal, self.first[non_terminal
            ]))
    print()

# 计算单个非终结符的first集
def calculate_first(self, grammar, non_terminal):
    if len(grammar.productions[non_terminal]) == 0:
```

```
        return

    for terminal in grammar.terminals:
        if terminal != "ε":
            self.first[terminal].add(terminal)

    for production in grammar.productions[non_terminal]:
        if production == "num":
            first_symbol = "num"
        elif len(production) > 1 and production[1] == "'":
            first_symbol = production[0:2]
        else:
            first_symbol = production[0]

        if first_symbol in grammar.terminals:
            self.first[non_terminal].add(first_symbol)
        elif first_symbol in grammar.non_terminals:
            self.calculate_first(grammar, first_symbol)
            self.first[non_terminal].update(self.first[first_symbol])

        if (
            first_symbol in grammar.non_terminals
            and "ε" in grammar.productions[first_symbol]
        ):
            if len(production) > 1:
                if first_symbol == production[0]:
                    second_symbol = production[1]
                elif len(production) > 2 and first_symbol == production
                    [0:2]:
                    second_symbol = production[2]
                    if len(production) > 3 and production[3] == "'":
                        second_symbol += "'"
                else:  # 右侧只有一个ε
                    continue
                self.calculate_first(grammar, second_symbol)
                self.first[non_terminal].update(self.first[second_symbol])
```

主要功能函数是compute_first，用于计算文法的 First 集合。它接受一个grammar参数，表示输入的文法。

在compute_first函数中，首先对所有非终结符和终结符进行初始化，将它们的 First 集合初始化为空集。然后依次处理每个非终结符，调用calculate_first函数进行计算。

calculate_first函数用于计算单个非终结符的 First 集合。它接受两个参数：grammar表示输入的文法，non_terminal表示当前要计算 First 集合的非终结符。

首先，如果当前非终结符的产生式为空，则直接返回。接下来，将所有终结符加入到它们自己的 First 集合中。

然后，对于当前非终结符的每个产生式，获取产生式的第一个符号作为`first_symbol`。

如果`first_symbol`是终结符，则将它添加到当前非终结符的 First 集合中。

如果`first_symbol`是非终结符，则递归调用`calculate_first`函数计算`first_symbol`的 First 集合，并将结果合并到当前非终结符的 First 集合中。

接下来，如果`first_symbol`是非终结符且它的产生式中包含空串（"ϵ"），则需要处理空串的情况。对于产生式右侧只有一个符号的情况，直接跳过。对于产生式右侧有两个以上符号的情况，获取第二个符号作为`second_symbol`，然后递归调用`calculate_first`函数计算`second_symbol`的 First 集合，并将结果合并到当前非终结符的 First 集合中。

2.2.3 计算 Follow 集

下面的方法实现了构建非终结符的 FOLLOW 集：

对于非终结符 B，检查所有右部包含 B 的产生式 $A \to \alpha B Y_1 Y_2 \dots Y_k$：

遍历每一个 Y_i，如果：

- Y_i 是终结符，则 B 的 FOLLOW 集中增加 Y_i，终止遍历；
- Y_i 是非终结符，B 的 FOLLOW 集并上 Y_i 的 FIRST 集中非空的部分。此后检查 Y_i 的 FIRST 集中是否包含 ε（即是否能推导出 ε），若不包含，则终止遍历。

如果遍历完 Y_k 并且 $A \neq B$，则 A 的 FOLLOW 集包含在 B 的 FOLLOW 集中。

```
# 计算follow集
def compute_follow(self, grammar):
    start_symbol = grammar.start_symbol

    for non_terminal in grammar.non_terminals:  # 初始化
        self.follow[non_terminal] = set()

    self.follow[start_symbol].add("$")

    while True:
        changed = False

        for non_terminal in grammar.non_terminals:
            for production in grammar.productions[non_terminal]:
                symbols = production

                for i in range(len(symbols)):
                    symbol = symbols[i]
                    if i != len(symbols) - 1 and symbols[i + 1] == "'":
                        symbol += "'"
                        i += 1
```

```
                if symbol in grammar.non_terminals:  # 当前符号是非终结
                    符
                    if i == len(symbols) - 1:  # 最后一个符号
                        original_set = self.follow[symbol].copy()
                        self.follow[symbol].update(
                            self.follow[non_terminal]
                        )  # follow[symbol] = follow[symbol] + follow[
                            non_terminal]
                        changed = changed or (
                            original_set != self.follow[symbol]
                        )

                    else:  # 不是最后一个符号
                        next_symbol = symbols[i + 1]
                        if i + 1 != len(symbols) - 1 and symbols[i + 2]
                            == "'":
                            next_symbol += "'"

                        if next_symbol in grammar.terminals:  # 下一个
                            符号是终结符,直接加入
                            if next_symbol not in self.follow[symbol]:
                                original_set = self.follow[symbol].copy
                                    ()
                                self.follow[symbol].add(next_symbol)
                                changed = changed or (
                                    original_set != self.follow[symbol]
                                )

                        else:  # 下一个符号是非终结符
                            if (
                                "ε" in self.first[next_symbol]
                            ):  # next_symbol的first集中有ε
                                original_set = self.follow[symbol].copy
                                    ()
                                self.follow[symbol].update(
                                    self.follow[non_terminal]
                                )  # follow[symbol] = follow[symbol] +
                                    follow[non_terminal]
                                changed = changed or (
                                    original_set != self.follow[symbol]
                                )
```

```
                                original_set = self.follow[symbol].copy()
                                self.follow[symbol].update(
                                    self.first[next_symbol] - {"ε"}
                                )  # follow[symbol] = follow[symbol] +
                                   first[next_symbol] - {ε}
                                changed = changed or (
                                    original_set != self.follow[symbol]
                                )

            if not changed:
                break

    for non_terminal in grammar.non_terminals:  # 输出follow集
        print("Follow({}) = {}".format(non_terminal, self.follow[
            non_terminal]))
    print()
```

首先，代码通过初始化将每个非终结符的 Follow 集初始化为空集。然后，将起始符号的 Follow 集设置为包含结束符号"$"。

接下来，代码进入一个循环，直到不再有改变为止。在每一次循环中，代码遍历文法的所有非终结符号。

对于每个非终结符号，代码遍历该非终结符号的每个产生式。产生式的右部可以包含终结符号和非终结符号。

对于产生式右部的每个符号，代码根据不同的情况进行处理。

当前符号是非终结符号：

- 如果该非终结符号是产生式的最后一个符号，则将该非终结符号的 Follow 集与当前非终结符号的 Follow 集合并。即，$follow[symbol] = follow[symbol] + follow[non\_terminal]$。
- 如果该非终结符号不是产生式的最后一个符号，则判断下一个符号。
  - 如果下一个符号是终结符号，直接将其加入该非终结符号的 Follow 集。
  - 如果下一个符号是非终结符号，根据下一个符号的 First 集进行处理：
    - 如果下一个符号的 First 集中包含空串"ϵ"，则将该非终结符号的 Follow 集与当前非终结符号的 Follow 集合并。
    - 将该非终结符号的 Follow 集与下一个符号的 First 集（去除空串"ϵ"）合并。

代码通过检查 Follow 集是否发生改变来判断是否需要继续进行迭代。如果 Follow 集没有发生改变，则退出循环。

2.2.4 是否满足 LL(1) 文法

判断文法是否是 LL(1) 文法，即检查每个产生式 $A \to \alpha|\beta$，需要满足：

- $\mathbf{FIRST}(\alpha) \cap \mathbf{FIRST}(\beta) = \varnothing$
- 如果 $A \to \varepsilon$，$\mathbf{FIRST}(\alpha) \cap \mathbf{FOLLOW}(A) = \varnothing$

```
# 检查是否是LL1文法
def is_LL1(self, grammar):
    for non_terminal, production in grammar.productions.items():
        if len(production) > 1:  # 一个非终结符对应多个产生式
            for p1 in production:  # First集合不相交
                for p2 in production:
                    if p1 != p2:
                        head1 = p1[0]
                        head2 = p2[0]
                        if head1 == "n":
                            head1 = "num"
                        if head2 == "n":
                            head2 = "num"
                        if (
                            len(self.first[head1].intersection(self.first[
                                head2]))
                            > 0
                        ):
                            print(
                                "First集合相交: First{} = {} and First{} =
                                    {}".format(
                                    head1,
                                    self.first[head1],
                                    head2,
                                    self.first[head2],
                                )
                            )
                            return False
            epsilon_index = []
            for index in range(len(production)):
                head = production[index][0]
                if head == "n":
                    head = "num"
                if "ε" in self.first[head]:
                    epsilon_index.append(index)
            if len(epsilon_index) != 0:  # First含ε, 其他First集合和左部符
                号Follow集合不相交
                for index in range(len(production)):
                    if index not in epsilon_index:
                        head = production[index][0]
                        if head == "n":
                            head = "num"
```

```
                        if (
                            len(
                                self.first[head].intersection(
                                    self.follow[non_terminal]
                                )
                            )
                            > 0
                        ):
                            print(
                                "First集合和Follow集合相交: First{} = {}
                                   and Follow{} = {}".format(
                                    head,
                                    self.first[head],
                                    non_terminal,
                                    self.follow[non_terminal],
                                )
                            )
                            return False
    return True
```

代码首先遍历文法的每个非终结符号和对应的产生式。对于每个非终结符号，如果它对应的产生式数量大于 1（即存在多个产生式），则需要检查是否满足 LL(1) 文法的条件。

1. 检查 First 集合是否相交：
   - 对于每对不同的产生式 p1 和 p2，取出它们的头部符号 head1 和 head2。
   - 如果头部符号是”n”，则将其替换为”num”。
   - 如果 first[head1] 和 first[head2] 的交集非空，表示存在相交的 First 集合，即不满足 LL(1) 文法的条件。输出相交的 First 集合，并返回 False 表示不是 LL(1) 文法。
2. 检查是否存在产生式的 First 集合含有空串”ϵ”：
   - 遍历每个产生式，如果产生式头部符号的 First 集合中包含空串”ϵ”，则记录该产生式的索引。
   - 如果存在记录的索引，表示存在产生式的 First 集合含有空串”ϵ”。
     - 对于每个不含有空串”ϵ” 的产生式，检查其头部符号的 First 集合和非终结符号的 Follow 集合是否相交。
     - 如果 first[head].intersection(follow[non_terminal]) 的交集非空，表示存在相交的 First 集合和 Follow 集合，即不满足 LL(1) 文法的条件。输出相交的 First 集合和 Follow 集合，并返回 False 表示不是 LL(1) 文法。

如果代码执行完所有检查，并没有返回 False，则表示文法是 LL(1) 文法，返回 True 表示是 LL(1) 文法。

2.2.5 算法 4.2 构造预测分析表

```
# 教材算法4.2
def compute_predictions_table(self, grammar):
    for production in grammar.productions:
        left_element = production
        for right_list in grammar.productions[production]:
            if right_list[0] in grammar.non_terminals:  # 右侧第一个符号是
                非终结符
                for a in self.first[right_list[0]]:  # 把A->a加入M[A,a]
                    self.predictions_table[left_element, a] = right_list
                if "ε" in self.first[right_list[0]]:
                    for b in self.follow[left_element]:  # 把A->a加入M[A,b]
                        self.predictions_table[left_element, b] =
                            right_list
            else:  # 右侧第一个符号是终结符
                if right_list == "num":
                    self.predictions_table[left_element, "num"] =
                        right_list
                else:
                    self.predictions_table[left_element, right_list[0]] =
                        right_list
                if "ε" == right_list[0]:  # 把A->a加入M[A,a]
                    for b in self.follow[left_element]:  # 把A->a加入M[A,b]
                        self.predictions_table[left_element, b] =
                            right_list

    # 错误处理加入分析表
    for non_terminal in grammar.non_terminals:
        for symbol in self.follow[non_terminal]:
            if (non_terminal, symbol) not in self.predictions_table:
                self.predictions_table[non_terminal, symbol] = "synch"

    # 输出预测分析表
    for terminal in grammar.terminals:
        if terminal == "ε":
            terminal = "$"
        print("\t{:10}".format(terminal), end="")
    print()
    for non_terminal in grammar.non_terminals:
        print(non_terminal, end="")
        for i in grammar.terminals:
            if i == "ε":
                i = "$"
            if (non_terminal, i) in self.predictions_table:
```

```
                print(
                    "\t{:10}".format(
                        non_terminal
                        + " -> "
                        + self.predictions_table[non_terminal, i]
                    ),
                    end="",
                )
            else:
                print("\t{:10}".format(" "), end="")
        print()
    print()
```

这段代码是用于构建预测分析表的算法，实现了教材算法 4.2 的逻辑。

首先，代码通过遍历文法的产生式来构建预测分析表。对于每个产生式，将产生式的左部作为`left_element`。

接下来，代码遍历该产生式的每个右部。如果右部的第一个符号是非终结符，则将该非终结符的 First 集中的每个终结符号加入预测分析表。即，对于每个终结符号`a`，将产生式`right_list`加入`predictions_table[left_element, a]`。

如果右部的第一个符号的 First 集中包含空串"ϵ"，则还需要将左部的 Follow 集中的每个终结符号加入预测分析表。即，对于每个终结符号`b`，将产生式`right_list`加入`predictions_table[left_element, b]`。

如果右部的第一个符号是终结符，则将该终结符号加入预测分析表。如果右部的第一个符号是"ϵ"，则将左部的 Follow 集中的每个终结符号加入预测分析表。

接下来，代码处理错误情况。对于每个非终结符号，遍历其 Follow 集中的每个终结符号，如果`(non_terminal, symbol)`不在预测分析表中，则将其加入，并将对应的值设置为"`synch`"，表示错误处理。

最后，代码输出预测分析表。按照表格形式输出每个终结符号，并在每个终结符号下面列出对应的产生式。如果预测分析表中没有对应的产生式，则输出空字符串。

2.2.6 算法 4.1 LL(1) 预测分析

```
# 教材算法4.1
def parse(self, grammar, input_string):
    print("{:30}".format("Stack"), "\t{:30}".format("Input"), "\tOutput")
    stack = ["$", grammar.start_symbol]
    ptr = 0
    error_count = 0
    while True:
        print(
            "{:30}".format("".join(stack)),
```

```
            "\t{:30}".format(input_string[ptr:] + "$"),
            end="",
        )

        if ptr == len(input_string):  # 输入串已经读完
            now_string = "$"
        elif input_string[ptr].isdigit():  # 输入串中的符号是数字
            now_string = "num"
        else:
            now_string = input_string[ptr]

        if ptr == len(input_string) and stack[-1] == "$":  # 分析完毕，退出
            print("\t\n\033[93mComplete!\033[0m", end="")
            if error_count == 0:
                print("\t\033[92mNo error\033[0m\n")
            else:
                print("\t\033[91m{} error(s)\033[0m\n".format(error_count))
            break
        element = stack[-1]  # 栈顶元素
        if element in grammar.terminals:  # 栈顶元素是终结符
            if (element == "num" and now_string.isdigit()) or (
                element == now_string
            ):  # 栈顶元素和输入串中的符号相同
                stack.pop()
                ptr += 1
                print("\t")
            else:  # 栈顶元素和输入串中的符号不同
                print(
                    "\t\033[91mError\033[0m: {} != {}".format(element,
                        now_string)
                )
                error_count += 1
                stack.pop()
        else:
            if (element, now_string) in self.predictions_table:
                stack.pop()
                production = self.predictions_table[element, now_string]
                if production == "synch":  # 预测分析表中对应的产生式是
                    synch
                    print(
                        "\t\033[91mError\033[0m: predictions_table[{}, {}]
                            is synch".format(
                            element, now_string
```

```
                    )
                )
                error_count += 1
            else:  # 预测分析表中有对应的产生式
                print("\t{} -> {}".format(element, production))
                if production != "ε":
                    re_production = self.pre_reversed(production)  # 反
                        转产生式
                    i = 0
                    while i < len(re_production):  # 将产生式反序入栈
                        if (
                            i < len(re_production) - 1
                            and re_production[i + 1] == "'"
                        ):
                            stack.append(re_production[i : i + 2])
                            i += 2
                        elif re_production[i : i + 3] == "num":
                            stack.append("num")
                            i += 3
                        else:
                            stack.append(re_production[i])
                            i += 1
        else:  # 预测分析表中没有对应的产生式
            print(
                "\t\033[91mError\033[0m: predictions_table[{}, {}] is
                    empty".format(
                    element, now_string
                )
            )
            error_count += 1
            ptr += 1
```

代码使用了一个栈来模拟语法分析过程。具体步骤如下：

1. 初始化栈，将终结符”$” 和文法的起始符号压入栈中。
2. 循环进行以下步骤，直到分析完成或出现错误：
   - 打印当前的栈内容、输入串和输出。
   - 检查是否分析完成，即输入串已经读完并且栈顶元素为终结符”$”。如果是，则打印”Complete!”，检查是否存在错误并输出相关信息，然后退出循环。
   - 获取当前输入符号：
     - 如果输入串已经读完（`ptr == len(input_string)`），设置当前符号为”$”。
     - 如果输入串中的符号是数字，则设置当前符号为”num”。
     - 否则，设置当前符号为输入串中的当前字符。

- 获取栈顶元素。
  - 如果栈顶元素是终结符：
    - 如果栈顶元素和当前符号相同（包括数字和”num” 的匹配），表示匹配成功，将栈顶元素弹出，指针向前移动一位。
    - 否则，表示匹配失败，输出错误信息，并将栈顶元素弹出。错误计数加 1。
  - 如果栈顶元素是非终结符：
    - 检查预测分析表（predictions_table）中是否存在对应的产生式。
    - 如果存在：
      - 弹出栈顶元素。
      - 获取对应的产生式。
      - 如果产生式为”synch”，表示在预测分析表中对应的产生式是”synch”，即发生了错误。输出错误信息，并将错误计数加一。
      - 否则，输出栈顶元素和对应的产生式，并将产生式反序入栈（从右向左）。
    - 如果不存在，表示在预测分析表中对应的产生式是空的，即发生了错误。输出错误信息，并将指针向前移动一位。错误计数加一。

这样，预测分析器会不断从栈中弹出符号进行匹配或扩展，直到分析完成或出现错误。

在每次循环中，代码会打印当前的栈内容、输入串和输出，以及相应的错误信息（如果有错误）。最后，根据是否存在错误，输出相应的完成信息。

3 使用说明

直接运行Gramma_LL1.py即可

下面是main函数中，每条语句的逐行分析

3.1 输入给定文法

在Gramma_LL1.py中，首先将给定文法添加到文法类中，便于后续分析

```
grammar = Grammar()
grammar.add_production("E", "E+T")
grammar.add_production("E", "E-T")
grammar.add_production("E", "T")
grammar.add_production("T", "T*F")
grammar.add_production("T", "T/F")
grammar.add_production("T", "F")
grammar.add_production("F", "(E)")
grammar.add_production("F", "num")
grammar.start_symbol = "E"

grammar.print_grammar()
```

```
print("Non-terminals:", grammar.non_terminals)
print("Terminals:", grammar.terminals)
print("Start symbol:", grammar.start_symbol)
print("Productions:", grammar.productions, "\n")
```

打印出的文法类 Grammar 的情况如下

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

Non-terminals: {'F', 'T', 'E'}
Terminals: {'num', ')', '+', '-', '(', '/', '*'}
Start symbol: E
Productions: {'E': ['E+T', 'E-T', 'T'], 'T': ['T*F', 'T/F', 'F'], 'F': ['(E)', 'num']}
```

3.2 消除左递归、左公因子

之后，需要对文法进行消除左递归，消除左公因子

```
print("消除左递归和消除左公因子后")
grammar.eliminate_left_recursion()
grammar.eliminate_left_factor()

grammar.print_grammar()

print("Non-terminals:", grammar.non_terminals)
print("Terminals:", grammar.terminals)
print("Start symbol:", grammar.start_symbol)
print("Productions:", grammar.productions, "\n")
```

输出消除左递归和消除左公因子后的文法

```
消除左递归和消除左公因子后
E -> TE'
T -> FT'
F -> (E) | num
T' -> *FT' | /FT' | ε
E' -> +TE' | -TE' | ε

Non-terminals: {'T', "E'", 'E', "T'", 'F'}
Terminals: {'num', 'ε', ')', '+', '-', '(', '/', '*'}
Start symbol: E
Productions: {'E': ["TE'"], 'T': ["FT'"], 'F': ['(E)', 'num'], "T'": ["*FT'", "/FT'", 'ε'], "E'": ["+TE'",
    "-TE'", 'ε']}
```

准备工作做完后，可以开始进行 LL1 分析首先构建 LL1 分析器，并构建 First 集和 Follow 集合

```
ll1_parser = LL1Parser()
ll1_parser.compute_first(grammar)
```

```
ll1_parser.compute_follow(grammar)
```

3.3 进行 LL(1) 分析

在进行 LL1 的预测分析前，我们需要检查文法是否满足 LL(1) 文法的条件，在满足 LL(1) 文法的条件下，才能构建预测分析表，并对给定输入进行分析

```
ll1_parser = LL1Parser()
ll1_parser.compute_first(grammar)
ll1_parser.compute_follow(grammar)
if ll1_parser.is_LL1(grammar):
    print("经检查，该文法是LL1文法\n")
    ll1_parser.compute_predictions_table(grammar)
    input_string = ["1+2", "1+2*(3-(4/0))", "1+2*/(3-4/0))"]
    for s in input_string:
        ll1_parser.parse(grammar, s)
else:
    print("经检查，该文法不是LL1文法\n")
```

如果是 LL(1) 文法，则输出预测分析表

```
经检查，该文法是LL1文法
    num         $           )           +           -           (           /           *
T   T -> FT'    T -> synch  T -> synch  T -> synch  T -> synch  T -> FT'
E'              E' -> ε     E' -> ε     E' -> +TE'  E' -> -TE'
E   E -> TE'    E -> synch  E -> synch                          E -> TE'
T'              T' -> ε     T' -> ε     T' -> ε     T' -> ε                 T' -> /FT'  T' -> *FT'
F   F -> num    F -> synch  F -> synch  F -> synch  F -> synch  F -> (E)     F -> synch  F -> synch
```

4 测试

```
input_string = ["1+2", "1+2*(3-(4/0))", "1+2*/(3-4/0))"]
```

4.1 测试 1+2

该测试集用于测试一个简单的算数表达式能否被正确识别

4.1.1 输出结果

```
Stack                       Input                       Output
$E                          1+2$                        E -> TE'
$E'T                        1+2$                        T -> FT'
$E'T'F                      1+2$                        F -> num
$E'T'num                    1+2$
$E'T'                       +2$                         T' -> ε
$E'                         +2$                         E' -> +TE'
$E'T+                       +2$
```

```
$E'T                          2$                          T -> FT'
$E'T'F                        2$                          F -> num
$E'T'num                      2$
$E'T'                         $                           T' -> ε
$E'                           $                           E' -> ε
$                             $
Complete!    No error
```

4.1.2 输出结果分析

这个 LL(1) 分析过程是针对输入表达式”1+2” 进行的。下面是对该分析过程的简要分析结果:

- 在开始时，我们有一个起始非终结符为 E 的文法规则。根据输入”1+2”，我们将推导出 E -> TE’。
- 然后，我们将推导 T -> FT’，因为输入中的下一个符号是”1”。
- 接着，我们将推导 F -> num，因为输入中的下一个符号是”1”，而”2” 是一个数字。
- 接下来，我们将进一步推导 E’ -> +TE’，因为输入中的下一个符号是”+”。
- 然后，我们将推导 T -> FT’，因为输入中的下一个符号是”2”。
- 接着，我们将推导 F -> num，因为输入中的下一个符号是”2”，而”2” 是一个数字。
- 然后，我们将推导出 T’ -> ε，因为输入已经结束，而 T’ 可以为空。
- 最后，我们将推导出 E’ -> ε，因为输入已经结束，而 E’ 可以为空。这表示我们成功地完成了对输入表达式的分析。

整个分析过程没有出现错误，意味着输入表达式是符合给定的文法规则的，并且可以被成功地分析和接受。

4.2 测试 1+2*(3-(4/0))

4.2.1 输出结果

```
Stack                         Input                       Output
$E                            1+2*(3-(4/0))$              E -> TE'
$E'T                          1+2*(3-(4/0))$              T -> FT'
$E'T'F                        1+2*(3-(4/0))$              F -> num
$E'T'num                      1+2*(3-(4/0))$
$E'T'                         +2*(3-(4/0))$               T' -> ε
$E'                           +2*(3-(4/0))$               E' -> +TE'
$E'T+                         +2*(3-(4/0))$
$E'T                          2*(3-(4/0))$                T -> FT'
$E'T'F                        2*(3-(4/0))$                F -> num
$E'T'num                      2*(3-(4/0))$
$E'T'                         *(3-(4/0))$                 T' -> *FT'
$E'T'F*                       *(3-(4/0))$
$E'T'F                        (3-(4/0))$                  F -> (E)
$E'T')E(                      (3-(4/0))$
$E'T')E                       3-(4/0))$                   E -> TE'
$E'T')E'T                     3-(4/0))$                   T -> FT'
$E'T')E'T'F                   3-(4/0))$                   F -> num
$E'T')E'T'num                 3-(4/0))$
```

```
$E'T')E'T'                      -(4/0))$                        T' -> ε
$E'T')E'                        -(4/0))$                        E' -> -TE'
$E'T')E'T-                      -(4/0))$
$E'T')E'T                       (4/0))$                         T -> FT'
$E'T')E'T'F                     (4/0))$                         F -> (E)
$E'T')E'T')E(                   (4/0))$
$E'T')E'T')E                    4/0))$                          E -> TE'
$E'T')E'T')E'T                  4/0))$                          T -> FT'
$E'T')E'T')E'T'F                4/0))$                          F -> num
$E'T')E'T')E'T'num              4/0))$
$E'T')E'T')E'T'                 /0))$                           T' -> /FT'
$E'T')E'T')E'T'F/               /0))$
$E'T')E'T')E'T'F                0))$                            F -> num
$E'T')E'T')E'T'num              0))$
$E'T')E'T')E'T'                 ))$                             T' -> ε
$E'T')E'T')E'                   ))$                             E' -> ε
$E'T')E'T')                     ))$
$E'T')E'T'                      )$                              T' -> ε
$E'T')E'                        )$                              E' -> ε
$E'T')                          )$
$E'T'                           $                               T' -> ε
$E'                             $                               E' -> ε
$                               $
Complete!    No error
```

4.2.2 输出结果分析

对于输入表达式”1+2*(3-(4/0))”，经过 LL(1) 分析，我们发现：

- 输入表达式符合给定的文法规则，没有语法错误。
- 分析过程中成功地推导出了输入表达式的每个符号，从起始非终结符 E 开始，按照产生式规则进行推导。
- 分析过程中遵循了 LL(1) 文法的规则，每次选择正确的产生式进行推导，并根据输入的下一个符号进行匹配。
- 最终，成功地完成了对输入表达式的分析，没有剩余的输入符号，且分析栈为空。

分析过程中没遇到了除以零的情况，这表明输入表达式在语义上是不正确的，但输入表达式在语法上是合法的。综上所述，这个 LL(1) 分析结果表明，这个表达式语法上无误

4.3 测试 1+2*/(3-4/0))

4.3.1 输出结果

```
Stack                           Input                           Output
$E                              1+2*/(3-4/0))$                  E -> TE'
$E'T                            1+2*/(3-4/0))$                  T -> FT'
$E'T'F                          1+2*/(3-4/0))$                  F -> num
$E'T'num                        1+2*/(3-4/0))$
$E'T'                           +2*/(3-4/0))$                   T' -> ε
$E'                             +2*/(3-4/0))$                   E' -> +TE'
$E'T+                           +2*/(3-4/0))$
$E'T                            2*/(3-4/0))$                    T -> FT'
```

```
$E'T'F                          2*/(3-4/0))$                    F -> num
$E'T'num                        2*/(3-4/0))$
$E'T'                           */(3-4/0))$                     T' -> *FT'
$E'T'F*                         */(3-4/0))$
$E'T'F                          /(3-4/0))$                      Error: predictions_table[F, /] is synch
$E'T'                           /(3-4/0))$                      T' -> /FT'
$E'T'F/                         /(3-4/0))$
$E'T'F                          (3-4/0))$                       F -> (E)
$E'T')E(                        (3-4/0))$
$E'T')E                         3-4/0))$                        E -> TE'
$E'T')E'T                       3-4/0))$                        T -> FT'
$E'T')E'T'F                     3-4/0))$                        F -> num
$E'T')E'T'num                   3-4/0))$
$E'T')E'T'                      -4/0))$                         T' -> ε
$E'T')E'                        -4/0))$                         E' -> -TE'
$E'T')E'T-                      -4/0))$
$E'T')E'T                       4/0))$                          T -> FT'
$E'T')E'T'F                     4/0))$                          F -> num
$E'T')E'T'num                   4/0))$
$E'T')E'T'                      /0))$                           T' -> /FT'
$E'T')E'T'F/                    /0))$
$E'T')E'T'F                     0))$                            F -> num
$E'T')E'T'num                   0))$
$E'T')E'T'                      ))$                             T' -> ε
$E'T')E'                        ))$                             E' -> ε
$E'T')                          ))$
$E'T'                           )$                              T' -> ε
$E'                             )$                              E' -> ε
$                               )$                              Error: predictions_table[$, )] is empty
$                               $
Complete!    2 error(s)
```

4.3.2 输出结果分析

这个输出结果显示了对输入表达式”1+2*/(3-4/0))” 进行 LL(1) 分析的过程。以下是对该输出结果的总结分析：

- 在分析开始时，初始栈中有起始符号 $E，并且输入中包含完整的表达式”1+2*/(3-4/0))”。
- 在推导过程中，栈中的符号逐步变化，同时消耗输入符号，直到最终栈和输入都为空。
- 在分析过程中，我们看到了两个错误。第一个错误发生在推导 F -> (E) 时，因为输入中的下一个符号是”/”，而在预测表中，F 和”/” 之间没有产生式，因此出现了错误。第二个错误发生在分析完成后，栈和输入都为空时，预测表中没有定义栈顶符号为”$” 且输入符号为”)” 的产生式，因此也出现了错误。
- 在第一个错误发生后，分析过程进行了错误恢复，将错误的输入符号视为同步点，弹出当前栈顶符号并继续分析。在第二个错误发生时，分析器会忽略多余的输入，”)” 被视为不属于语法规则的一部分，向前移动向前指针，跳过该符号

总体来说，这个输出结果显示了分析过程中的错误情况，但也展示了错误恢复的能力，使得分析过程能够继续进行直到完成。综上所述，这个输出结果表明在对输入表达式进行 LL(1) 分析的过程中，存在两个错误，并且分析过程能够进行错误恢复以继续分析，最终完成了分析过程。

5 实验总结

在本次实验中，我编写了一个 LL(1) 语法分析程序，通过这个实验我对语法分析的流程有了更清晰的理解，并加深了对相关知识点的掌握。

我设计的语法分析程序的架构比较简单，主要的难点在于算法的设计。在实现过程中，我使用了 Python 的语法特性，这使得维护文法产生式和预测分析表等变得更容易，大大减少了编程的复杂度。

然而，我也意识到我的语法分析程序仍然存在许多待改进的地方。例如，当出现分析错误时，程序可以输出更多的报错信息，以帮助用户更好地理解错误的原因。另外，在进行文法转换和求解集合时，程序的鲁棒性还有待提高，以处理各种边界情况和异常输入。尽管由于时间限制，我无法一一考虑到所有可能的情况，但我认识到这些改进是提高程序质量的重要一步。

通过这次实验，我不仅对课内知识有了更深入的认识，还提高了我的 Python 编程能力。我从中获益匪浅，对语法分析和算法设计有了更深入的理解，并学会了如何运用所学知识解决实际问题。

总之，这次实验是一次有益的经验，我很高兴能够完成并取得一定的成果。我期待在以后的学习和实践中继续深入研究语法分析以及相关领域的知识。