《编译原理》

实验报告

项目名称 实验二《预测分析法设计与实现》

专业班级 软件 1805 班

学　　号 8209180516

姓　　名 陈冉飞

实验成绩：

批阅教师：

2020 年 5 月 22 日

实验 2《预测分析法设计与实现》

实验学时：_____ 实验地点：_____ 实验日期：_____

一、实验目的

加深对语法分析器工作过程的理解；加强对预测分析法实现语法分析程序的掌握；能够采用一种编程语言实现简单的语法分析程序；能够使用自己编写的分析程序对简单的程序段进行语法翻译。

二、实验内容

用预测分析法编制语法分析程序，语法分析程序的实现可以采用任何一种编程语言和工具。

三、实验方法

1、项目利用 Github 进行项目整体管理，完整代码可见文档最下边或者 GitHub 仓库链接(已同步至最新版本)https://github.com/Aczy156/Compiling-Principle-Work

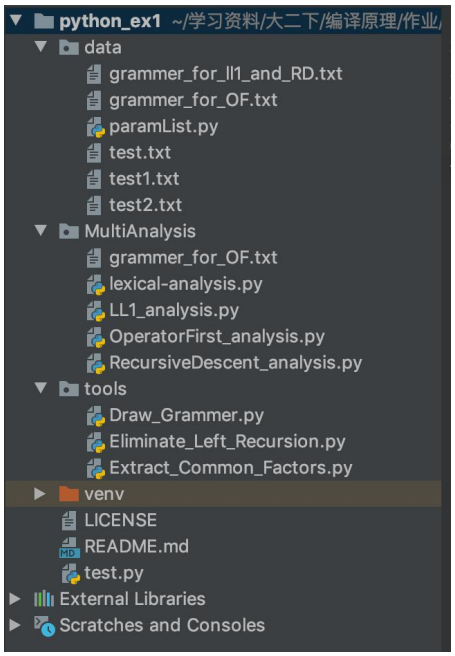
2、项目利用 Python(python 3)进行项目开发，开发 IDE 为 pycharm，项目内包含实验二、实验三、实验四，然后将整个项目模块化管理(见下图，符合较好的项目高内聚低耦合)，其中

- data
  - Grammer_for_ll1_and_RD.txt，是存放用于实验二预测分析法和实验三递归下降分析文法的测试文法
  - Grammer_for_OF.txt，是存放用于实验四算符优先文法的测试文法。
- MultiAnalysis
  - LL1_analysis.py 是 LL1 分析法的整体过程。
  - RecursiveDescent_analysis.py 是递归下降分析的整体过程。
  - OperatorFirst_analysis.py 是算符优先文法的整体分析过程。
- tools[在 MultiAnalysis 中的各种文法中多次使用，所以提取出来子模块，放进工具中来模块化管理]
  - Draw_Grammer.py，利用 python 中 prettytable 来专门做文法的可视化。
  - Eliminate_Left_Recursion.py，利用消除左递归算法来专门处理对于自上而下的文法(例如实验二中的预测分析文法和实验三的递归下降文法)的左递归的问题。
  - Extract_Common_Factors.py，利用 LCP(最长公共前缀)来提取公因子并消除。

3、数据结构(命名基本保持一致，除了在一些情境中要利用 python deepcopy 深备份一份或者要进行更改会命名为 new_grammer/new_vn 等等)：

- 文法 grammer：文法在通过预处理过后，通过 python 中的数据结构 dictionary(dict、字典)来映射，例如{'E': ['E+T', 'T'], 'T': ['T*F', 'F'], 'F': ['(E)', 'i']}，用来表示通过数据预处理之后的最基本的算术表达式文法。

- 非终结符 vn：非终结符通过 list 来盛放。例如['E', 'T', 'F']，用来表示通过数据预处理之后的最基本的算术表达式文法的非终结符。
- 终结符 vt：终结符通过 list 来盛放。例如 ['(', ')', 'i', '+', '*', '#']，用来表示通过数据预处理之后的最基本的算术表达式文法的非终结符。



4、由于项目模块化开发，并没有严格的按照实验二、实验三、实验四这样每个实验自己单独有一个独立的项目或者程式段，所以并没有把所有代码完全粘入，只是依据每个算法的所需的模块来详细粘入。(完整代码 https://github.com/Aczy156/Compiling-Principle-Work)

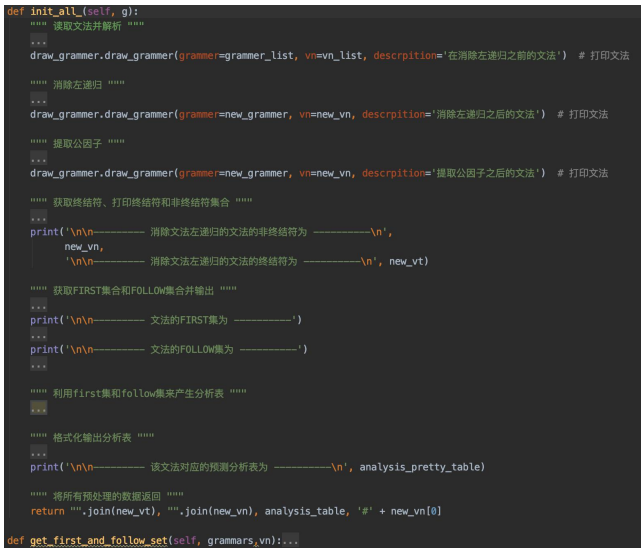
四、实验步骤

(一)程序的整体结构(详细的步骤在下边逐步展开)(代码已折叠)

- init 为对象初始化传参数，将从 txt 中读取到的文法传过来。
- init_all_包括所有的 LL1 文法的初始化工作，包括对文法读取、消除左递归、提取公因子、输出非终结符 vn 终结符 vt、得到并输出 FIRST 集 FOLLOW 集、得到分析表、输出分析表等，然后最后统一返回对象的对应参数，用于下边的模拟 LL1 的过程。
- Get_first_and_follow_set 用于获得 FIRST 集和 FOLLOW 集。
- LL1_analysis_solve 用于模拟 LL1 分析的过程

```python
from prettytable import PrettyTable
import re
from tools.Eliminate_Left_Recursion import EliminateLeftRecursion
from tools.Extract_Common_Factors import ExtractCommonFactors
import tools.Draw_Grammer as draw_grammer


class LL1_analysis:
    def __init__(self, Gram):
        self.vt, self.vn, self.analysis_table, self.stack_str = self.init_all_(g=Gram)
        self.ptr = 0

    def init_all_(self, g):...

    def get_first_and_follow_set(self, grammars,vn):...

    """ LL1分析过程 """
    def LL1_analysis_solve(self, goal_str, ans_table):...


if __name__ == '__main__':
    """ main 1输入文法 2输入要分析的字符串 """
    grammer = str(open('./data/grammer_for_ll1_and_RD.txt').read())
    ll1_analysis = LL1_analysis(Gram=grammer)
    while True:
        goal_str = str(input('请输入字符串(exit跳出循环):')) + '#'
        if goal_str == 'exit':
            break
        else:
            ans_table = PrettyTable(['分析栈', '输入串', '操作'])
            ll1_analysis.LL1_analysis_solve(goal_str=goal_str, ans_table=ans_table)
            print(ans_table)
```

其中的 init_all 模块的结构(代码已折叠，详细过程在下边)



```
def init_all_(self, g):
    """ 读取文法并解析 """
    ...
    draw_grammer.draw_grammer(grammer=grammer_list, vn=vn_list, descrpition='在消除左递归之前的文法')  # 打印文法

    """ 消除左递归 """
    ...
    draw_grammer.draw_grammer(grammer=new_grammer, vn=new_vn, descrpition='消除左递归之后的文法')  # 打印文法

    """ 提取公因子 """
    ...
    draw_grammer.draw_grammer(grammer=new_grammer, vn=new_vn, descrpition='提取公因子之后的文法')  # 打印文法

    """ 获取终结符、打印终结符和非终结符集合 """
    ...
    print('\n\n---------- 消除文法左递归的文法的非终结符为 ----------\n',
          new_vn,
          '\n\n---------- 消除文法左递归的文法的终结符为 ----------\n', new_vt)

    """ 获取FIRST集合和FOLLOW集合并输出 """
    ...
    print('\n\n---------- 文法的FIRST集为 ----------')
    ...
    print('\n\n---------- 文法的FOLLOW集为 ----------')
    ...

    """ 利用first集和follow集来产生分析表 """
    ...

    """ 格式化输出分析表 """
    ...
    print('\n\n---------- 该文法对应的预测分析表为 ----------\n', analysis_pretty_table)

    """ 将所有预处理的数据返回 """
    return "".join(new_vt), "".join(new_vn), analysis_table, '#' + new_vn[0]

def get_first_and_follow_set(self, grammars,vn):...
```

其中的 get_first_and_follow_set 模块的结构(代码已折叠，详细过程在下边)

```
def get_first_and_follow_set(self, grammars,vn):
    FIRST = {}
    FOLLOW = {}
    """ first集和follow集初始化 """
    for str in grammars:...

    """ 获取first集 """
    for str in grammars:...
    for i in range(len(vn)):...

    """ 获取follow集合 """
    for i in range(len(vn)):...
    return FIRST, FOLLOW

""" LL1分析过程 """
def LL1_analysis_solve(self, goal_str, ans_table):...
```

(二)详细过程

①读入文法并进行文法预处理(包括清除空格、字符调整等等)，然后最后调用 tools 中的 Draw_Grammer 来可视化当前的文法。

Code

利用 python re 进行分割从而进行文法预处理

```
""" 读取文法并解析 """
grammer_list = {}  # grammer 字典，对于特定的非终结符 vn 来来映射产生式
vn_list = []  # 非终结符列表
for line in re.split('\n', g):
    # 清楚空格
    line = "".join([i for i in line if i not in ['', ' ']])
    if '->' in line:
        if line.split('->')[0] not in vn_list:
            vn_list.append(line.split('->')[0])
        for i in line.split('->')[1].split('|'):
            if grammer_list.get(line.split('->')[0]) is None:
                grammer_list[line.split('->')[0]] = []
```

```
            grammer_list[line.split('->')[0]].append(i)
        else:
            grammer_list[line.split('->')[0]].append(i)  # 用于消除左递归的字典填充
draw_grammer.draw_grammer(grammer=grammer_list, vn=vn_list, descrpition='在消除左递归之前的文法')  # 打印文法
```

Draw_Grammer.py

```
from prettytable import PrettyTable


def draw_grammer(grammer, vn,descrpition):
    advanced_grammar = PrettyTable(['编号', '箭头左边', '箭头右边', '产生式'])  # 利用 prettytable 来渲染出新的消去左递归的文法
    idx = 1
    for i in vn:
        for j in grammer[i]:
            advanced_grammar.add_row([idx, i, j, i + '->' + j])
            # only_grammer.append(i + '->' + j)
            idx += 1
    print('\n\n--------- '+descrpition+' ----------\n', advanced_grammar)
```

Output

```
--------- 在消除左递归之前的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |   E+T    | E->E+T |
|  2   |    E     |    T     |  E->T  |
|  3   |    T     |   T*F    | T->T*F |
|  4   |    T     |    F     |  T->F  |
|  5   |    F     |   (E)    | F->(E) |
|  6   |    F     |    i     |  F->i  |
+------+----------+----------+--------+
```

(复制粘贴格式有点跑偏，下图为 prettytable 来进行文法可视化的最终呈现样式)

②消除左递归

若文法中有形如 P→Pα的产生式，称为直接左递归，如 A→Ab|a，这类情况需要进行消除左递归。当文法 G 不含回路，也不含 ε 产生式，则下列算法可消除左递归。

大致过程如下：

①把文法 G 的非终结符按任意顺序排列成 P1,⋯,Pn

②for i:=1 to n do

for j:=1 to i-1 do

把形如 Pi → P j γ 规则改写成 P i → δ 1 | ... | δ k γ， 其中 P j → δ 1 | ... | δ k 是关于 P j 的全部规则;

消除关于 Pi 的直接左递归

③化简由②得到的文法（取消无用非终结符产生式）

即可消除左递归

Code

通过创建 tools 中的 Eliminate_Left_Recursion.py 中 EliminateLeftRecursion 这个类的对象来执行其中的 remove_left_recursion 函数(利用对象初始化时传入的 grammer 和非终结符 vn 来进行消去)来对 grammer 进行消除左递归，并返回消除左递归后的新的 grammer 和 vn，并用 prettytable 来进行可视化文法

```
""" 消除左递归 """
# print('==========', grammer_list, '+++++++', vn_list)
eliminate_left_recursion = EliminateLeftRecursion(grammer=grammer_list,
vn=vn_list)
new_grammer, new_vn = eliminate_left_recursion.remove_left_recursion()
draw_grammer.draw_grammer(grammer=new_grammer, vn=new_vn, descrpition='消除左
递归之后的文法')  # 打印文法
```

Eliminate_Left_Recursion.py

```
import copy


class EliminateLeftRecursion:
    def __init__(self, grammer, vn):
        self.grammer = grammer
        self.vn = vn
        # 非终结符中用于增添的可选择的大写字母
        self.replace = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T',
```

```
                'U', 'V', 'W', 'X', 'Y', 'Z']

    def remove_left_recursion(self):
        """ 消除左递归 """
        new_grammer = copy.deepcopy(self.grammer)
        new_ac_set = copy.deepcopy(self.vn)
        for i in range(len(self.vn)): # 利用两层循环来消除左递归
            for j in range(0, i):
                new_grammer = self.convert(self.vn[i], self.vn[j], new_grammer)
            new_grammer, new_ac_set = self.clean_direct_recur(self.vn[i],
new_grammer, new_ac_set)
        return new_grammer, new_ac_set

    def convert(self, ch_i, ch_j, grammer):
        """ 对特定的字符进行转换 """
        rules = copy.deepcopy(grammer)  # 复制一份
        for key in grammer.keys():
            for item_i in grammer[key]:
                if ch_i == key and ch_j == item_i[0]:
                    rules[key].remove(item_i)
                    for item_j in grammer[ch_j]:
                        rules[key].append(item_j + item_i[1:])
        return rules

    def clean_direct_recur(self, ch_i, grammer, new_ac_set):
        """ 清除直接左递归 """
        ch = ''
        flag = 0
        rules = copy.deepcopy(grammer)

        for temp in self.replace: # 选择未被使用的非终结符
            if temp not in new_ac_set:
                ch = temp
                break

        for key in grammer.keys():
            for item_i in grammer[key]:
                if ch_i == key and ch_i == item_i[0]:
                    flag = 1
                    if ch not in rules.keys():
                        rules[ch] = []

                    rules[ch].append(item_i[1:] + ch)
                    rules[key].remove(item_i)
```

```
        if flag == 0:  # 不存在左递归，不用消去
            return rules, new_ac_set

        for key in grammer.keys():
            for item_i in grammer[key]:
                if ch_i == key and ch_i != item_i[0]:
                    if ch not in rules.keys():
                        rules[ch] = []
                    rules[ch_i].append(item_i + ch)
                    rules[key].remove(item_i)
        rules[ch].append('ε')  # 空输入在最后，不会影响递归下降
        new_ac_set.append(ch)
        # print(new_ac_set)
        # print(rules)
        return rules, new_ac_set
```

Output

```
---------- 消除左递归之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
|  8   |    B     |    ε     |  B->ε  |
+------+----------+----------+--------+
```

(复制粘贴格式有点跑偏，下图为 prettytable 来进行文法可视化的最终呈现样式)

```
---------- 消除左递归之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
|  8   |    B     |    ε     |  B->ε  |
+------+----------+----------+--------+
```

③提取公因子

若文法中有形如 S → aB1|aB2|aB3|aB4|...|aBn|y，此时前 n 项有共同的因子 a，这类情况需要利用 LCP 进行提取公因子，然后再拼接来消除公因子，得到 S → aS'|y，S'→ B1|B2|B3|...|Bn 这样的形式。

Code

通过创建 tools 中的 Extract_Common_Factors.py 中 ExtractCommonFactors 这个类的对象来执行其中的 remove_common_factor 函数(利用对象初始化时传入的 grammer 和非终结符 vn 来进行消去)来对 grammer 进行提取公因子, 并返回提出公因子并处理后的新的 grammer 和 vn, 并用 prettytable 来进行可视化文法。

```
""" 提取公因子 """
extractcommonfactors = ExtractCommonFactors(grammer=new_grammer, vn=new_vn)
new_grammer, new_vn = extractcommonfactors.remove_common_factor()
# print('==========', new_grammer, '+++++++', new_vn)
draw_grammer.draw_grammer(grammer=new_grammer, vn=new_vn, descrpition='提取公因子之后的文法')  # 打印文法
```

Extract_Common_Factors.py

```
import random


class ExtractCommonFactors:
    def __init__(self, grammer, vn):
        self.grammer = grammer
        self.vn = vn

    def LCP(self, i, j, rules):  # 获取的最长公共前缀
        """ LCP 获取两个字符串之间最长公共前缀 """
        strs = [rules[i], rules[j]]
        res = ''
        for each in zip(*strs):
            if len(set(each)) == 1:
                res += each[0]
            else:
                return res
        return res

    def get_lcp_res(self, key): #获得每个拥有公共前缀的元素下标
```

```
    """ 获得LCP公共前缀的索引 """
    res = {}
    rules = self.grammer[key]
    for i in range(len(rules)):
        for j in range(i+1, len(rules)):
            temp = self.LCP(i,j,rules)
            if temp not in res.keys():
                res[temp] = set()
            res[temp].add(i)
            res[temp].add(j)
    if '' in res.keys():
        res.pop('')
    return res

def remove_common_factor(self):
    keys = list(self.grammer.keys())  # 事先保存好没有修改过的grammer_key
    for key in keys:
        while (True):
            res = self.get_lcp_res(key)
            if (res == {}):  # 不断迭代，直到没有公共前缀
                break
            dels = []  # 存储需要删除的符号串
            lcp = list(res.keys())[0]  # 策略：每次取一个公共前缀
            ch = ''
            while (True):
                temp = chr(random.randint(65, 90))
                if temp not in self.vn:
                    ch = temp
                    break
            self.vn.append(ch)
            for i in res[lcp]:  # res[lcp]存储的要消除公共因子的元素下标
                string = self.grammer[key][i]
                dels.append(string)
                string = string.lstrip(lcp)
                if string == '':
                    string += '#'
                if ch not in self.grammer.keys():
                    self.grammer[ch] = []
                self.grammer[ch].append(string)  # 加到新的产生式里面
            for string in dels:  # 从原来的产生式里面删除
                self.grammer[key].remove(string)
            self.grammer[key].append(lcp + ch)
    return self.grammer, self.vn
```

Output

```
--------- 提取公因子之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
|  8   |    B     |    ε     |  B->ε  |
+------+----------+----------+--------+
```

(复制粘贴格式有点跑偏，下图为 prettytable 来进行文法可视化的最终呈现样式)

④构造 FIRST 集和 FOLLOW 集合

FIRST 集构造的三个规则

1、若 X ∈VT，则 FIRST(X)={X}

2、若 X∈VN，且有产生式 X→a…，则把 a 加入到 FIRST(X)中；若 X→ε也是一条产生式，则把 ε 也加到 FIRST(X)中。

3、若 X→Y…是一个产生式且 Y∈VN, 则把 FIRST(Y)中的所有非ε元素都加到 FIRST(X)中；若 X → Y1Y2…YK 是一个产生式，Y1,Y2,…,Y(i-1)都是非终结符，而且，对于任何 j,1≤j ≤i-1, FIRST(Yj)都含有ε (即 Y1..Y(i-1) ε)，则把 FIRST(Yj)中的所有非ε元素都加到 FIRST(X)中；特别是，若所有的 FIRST(Yj , j=1,2,…,K)均含有ε，则把 ε 加到 FRIST(X)中。

FOLLOW 集构造的三个规则

1、对于文法的开始符号 S，置#于 FOLLOW(S) 中；

2、若 A→α B β是一个产生式，则把 FIRST(β)\{ε}加至 FOLLOW(B)中；

3、若 A→α B 是一个产生式，或 A→ αBβ是 一个产生式而β ε (即 ε FIRST(β))， 则把 FOLLOW(A)加至 FOLLOW(B)中。

Code

通过多次扫描来将无限空间的遍历缩小为优先空间，就需要在每一次 FIRST 集/FOLLOW 集进行刷新之后(就算只有一个非终结符的 FIRST 集/FOLLOW 集发生了变化)，都要重新进行扫描，考虑到最极端的情况每一次只自下而上最底层的更新一次，那么最少需要 len(vn)次数的遍历即 `for i in range(len(vn)):`可满足。

```
""" 获取 FIRST 集合和 FOLLOW 集合并输出 """
FIRST, FOLLOW = self.get_first_and_follow_set(grammars=only_grammer, vn=new_vn)
# print(FIRST)
# print(FOLLOW)
print('\n\n---------- 文法的 FIRST 集为 ----------')
for i, j in FIRST.items():
    str = j[0]
    for temp in j[1:]:
        str = str + ',' + temp
    print("FIRST(" + i + ")" + " = {" + str + "}")
print('\n\n---------- 文法的 FOLLOW 集为 ----------')
for i, j in FOLLOW.items():
    str = j[0]
    for temp in j[1:]:
        str = str + ',' + temp
    print("FOLLOW(" + i + ")" + " = {" + str + "}")
```

Get_first_and_follow_set 方法

```
def get_first_and_follow_set(self, grammars,vn):
    FIRST = {}
    FOLLOW = {}
    """ first 集和 follow 集初始化 """
    for str in grammars: # 初始化 first 集 和 follow 集合字典的键值对中的 值 为空
        part_begin = str.split("->")[0]
        part_end = str.split("->")[1]
        FIRST[part_begin] = ""
        FOLLOW[part_begin] = "#"

    """ 获取 first 集 """
    for str in grammars: # 求 first 集 中第第一部分针对 ->  直接推出第一个字符为终结符 部分
```

```
        part_begin = str.split("->")[0]
        part_end = str.split("->")[1]
        if not part_end[0].isupper():
            FIRST[part_begin] = FIRST.get(part_begin) + part_end[0]
    for i in range(len(vn)):
        while True:
            test = FIRST
            for str in grammars: # 求first第二部分 针对 A -> B型  把B的first集加到A 的first集合中
                part_begin = ''
                part_end = ''
                part_begin += str.split('->')[0]
                part_end += str.split('->')[1]
                ##如果型如A ->B 则把B的first集加到A 的first集中去
                if part_end[0].isupper():
                    FIRST[part_begin] = FIRST.get(part_begin) +
FIRST.get(part_end[0])

            # 去除重复项
            for i, j in FIRST.items():
                temp = ""
                for word in list(set(j)):
                    temp += word
                FIRST[i] = temp
            if test == FIRST:
                break

    """ 获取follow集合 """
    for i in range(len(vn)):
        while True:
            test = FOLLOW
            # 计算follow集的第一部分，先计算 S -> A b 类型的
            for str in grammars:
                part_begin = str.split("->")[0]
                part_end = str.split("->")[1]
                ##如果是 S->a 直接推出终结符 则 continue
                if len(part_end) == 1:
                    continue
                ##否则执行下面的操作
                else:
                    # 将->后面的分开再倒序
                    temp = []
                    for i in part_end:
                        temp.append(i)
```

```
                temp.reverse()
                # 如果非终结符在句型的末端则把"#" 加入进去
                if temp[0].isupper():
                    FOLLOW[temp[0]] = FOLLOW.get(temp[0]) +
FOLLOW.get(part_begin)
                    temp1 = temp[0]
                    for i in temp[1:]:
                        if not i.isupper():
                            temp1 = i
                        else:
                            if temp1.isupper():
                                FOLLOW[i] = FOLLOW.get(i) +
FIRST.get(temp1).replace("ε", "")
                                if ('ε' in FIRST.get(temp1)):
                                    FOLLOW[i] = FOLLOW.get(i) + FOLLOW.get(part_begin)
                                else:
                                    FOLLOW[i] = FOLLOW.get(i) + temp1
                                temp1 = i
                # 如果终结符在句型的末端
                else:
                    temp1 = temp[0]
                    for i in temp[1:]:
                        if not i.isupper():
                            temp1 = i
                        else:
                            if temp1.isupper():
                                FOLLOW[i] = FOLLOW.get(i) + FIRST.get(temp1)
                            else:
                                FOLLOW[i] = FOLLOW.get(i) + temp1
                            temp1 = i

            ##去除重复项
            for i, j in FOLLOW.items():
                temp = ""
                for word in list(set(j)):
                    temp += word
                FOLLOW[i] = temp
            if test == FOLLOW:
                break
    return FIRST, FOLLOW
```

Output

```
--------- 文法的 FIRST 集为 ----------
FIRST(E) = {i,(}
```

```
FIRST(T) = {i,(}
FIRST(F) = {i,(}
FIRST(A) = {+,ε}
FIRST(B) = {*,ε}



---------- 文法的 FOLLOW 集为 ----------
FOLLOW(E) = {#,)}
FOLLOW(T) = {#,),+}
FOLLOW(F) = {#,*,),+}
FOLLOW(A) = {#,)}
FOLLOW(B) = {#,),+}
```

⑤利用 FIRST 集和 FOLLOW 集合构造分析表

1、对文法 G 的每个产生式 A→α执行第 2 步和第 3 步；

2、对每个终结符 a∈FIRST(α)，把 A→α 加至 M[A,a]中；

3、若ε∈FIRST(α)，则对任何 b∈FOLLOW(A) ， 把 A→α 加至 M[A,b]中；

4、把所有无定义的 M[A,a]标上“出错标志”。

Code

利用 FIRST 集合和 FOLLOW 集合，根据上面的构造的四条规则(error 出错标志用 None 来替代)来获取分析表，然后通过 prettytable 进行输出

```
""" 利用 first 集和 follow 集来产生分析表 """
analysis_table = [[None] * (1 + len(new_vt)) for row in range(1 + len(new_vn))]
analysis_table[0][0] = ' '
for i in range(len(new_vt)):
    analysis_table[0][i + 1] = new_vt[i]
for i in range(len(new_vn)):
    analysis_table[i + 1][0] = new_vn[i]
    for t in new_grammer[new_vn[i]]:  # 遍历该文法的所有产生式
        if t == 'ε':  # 如果是ε 对应在 follow(i)中填上产生式
            for j in range(len(new_vt)):  # 遍历所有的终结符，并在对应的位置添加上对应的产生式子
                if new_vt[j] in FOLLOW[new_vn[i]]:  # Follow 为当前的非终结符的 follow 集
                    analysis_table[i + 1][j + 1] = 'ε'
        else:
            for j in range(len(new_vt)):
                if new_vt[j] in FIRST[new_vn[i]]:
                    analysis_table[i + 1][j + 1] = t
```

利用 prettytable 来打印

```
""" 格式化输出分析表 """
# print(analysis_table)
pretty_table_title = ['非终结符']
for i in new_vt:
    pretty_table_title.append(i)
analysis_pretty_table = PrettyTable(pretty_table_title)
for i in range(len(analysis_table) - 1):
    analysis_pretty_table.add_row(analysis_table[i + 1])
print('\n\n---------- 该文法对应的预测分析表为 ----------\n',
analysis_pretty_table)
```

Output

```
---------- 该文法对应的预测分析表为 ----------
 +----------+------+------+------+------+------+------+
| 非终结符 |  (   |  )   |  i   |  +   |  *   |  #   |
+----------+------+------+------+------+------+------+
|    E     |  TA  | None |  TA  | None | None | None |
|    T     |  FB  | None |  FB  | None | None | None |
|    F     |  i   | None |  i   | None | None | None |
|    A     | None |  ε   | None | +TA  | None |  ε   |
|    B     | None |  ε   | None |  ε   | *FB  |  ε   |
+----------+------+------+------+------+------+------+
```

(复制粘贴格式有点跑偏，下图为 prettytable 来进行文法可视化的最终呈现样式)

```
---------- 该文法对应的预测分析表为 ----------
 +----------+------+------+------+------+------+------+
| 非终结符 |  (   |  )   |  i   |  +   |  *   |  #   |
+----------+------+------+------+------+------+------+
|    E     |  TA  | None |  TA  | None | None | None |
|    T     |  FB  | None |  FB  | None | None | None |
|    F     |  i   | None |  i   | None | None | None |
|    A     | None |  ε   | None | +TA  | None |  ε   |
|    B     | None |  ε   | None |  ε   | *FB  |  ε   |
+----------+------+------+------+------+------+------+
```

⑥对于输入的制定字符串进行语法预测分析

预测分析程序按照 stack 栈顶符号 X 和当前输入符号 a 来做判断

1、若 X=a=#，则分析成功，程序结束

2、若 X=a≠#，则把 X 弹栈，指针后移，指向 a 的下一个

3、若 X 属于非终结符 vn，则查分析表

a) 如果 M[X,a]为某个候选式，然后弹出 X，并把这个候选式反序压栈

b) 如果 M[X,a]=ε，则翻出 X 即可。

c) 如果 M[X,a]为报错，则执行报错程序即可

Code 在模拟过程中利用字串来模拟栈，方便在打印整体分析过程的时候的输出。然后如

果反序压栈的话也只需要通过 str->list->利用 list 的 reverse()->join 为 str->加到原来的串即可

```
def LL1_analysis_solve(self, goal_str, ans_table):
    vt, vn, analysis_table, stack_str, ptr = self.vt, self.vn, self.analysis_table, self.stack_str, self.ptr
    while ptr >= 0 and ptr <= len(goal_str):
        stack_top = stack_str[len(stack_str) - 1]  # 获取栈顶
        goal_pos = goal_str[ptr]
        print(stack_str, '  ', stack_top, '  ', goal_pos)
        if (stack_top not in vt and stack_top not in vn) or goal_pos not in vt:  # 非法输入的情况
            print('输入不合法')
            return
        elif stack_top == goal_pos:
            if stack_top == '#':  # 栈顶符号=当前输入符号=#
                print('分析成功')
                ans_table.add_row([stack_str, goal_str[ptr:len(goal_str)], '分析成功'])
                return
            else:  # 栈顶符号=当前输入符号但是并不都等于#
                # --printstat 相等弹栈，指针前移
                print('栈顶符号=当前输入符号，指针前移')
                ans_table.add_row([stack_str, goal_str[ptr:len(goal_str)], '栈顶符号=当前输入符号，指针前移'])
                stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                ptr += 1
                continue
        lookup_table = analysis_table[max(vn.find(stack_top), vt.find(stack_top)) + 1][
            max(vn.find(goal_pos), vt.find(goal_pos)) + 1]
        print(stack_top, ' ~~~~~~~~~~ ', goal_pos, ' ~~~~~~~~~~ ', lookup_table)
        if lookup_table is not None:
            if lookup_table == 'ε':
                print('ε,弹栈')
                ans_table.add_row([stack_str, goal_str[ptr:len(goal_str)], 'ε,弹栈'])
                stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                continue
            else:
                print('存在对应的产生式并反向压栈')
                ans_table.add_row([stack_str, goal_str[ptr:len(goal_str)], '存在对应的产生式并反向压栈'])
                stack_str = stack_str[0:len(stack_str) - 1]  # 弹栈
                lt_list = list(lookup_table)
```

```
            lt_list.reverse()
            stack_str += "".join(lt_list)
            continue
        else:
            print('分析失败')
            return
```

#Output

```
请输入字符串(exit 跳出循环):i+i*i
分析成功
+---------+---------+----------------------------------+
| 分析栈 | 输入串 |               操作               |
+---------+---------+----------------------------------+
|   #E    | i+i*i# |    存在对应的产生式并反向压栈    |
|  #AT    | i+i*i# |    存在对应的产生式并反向压栈    |
|  #ABF   | i+i*i# |    存在对应的产生式并反向压栈    |
|  #ABi   | i+i*i# | 栈顶符号=当前输入符号，指针前移 |
|  #AB    | +i*i#  |             ε,弹栈              |
|   #A    | +i*i#  |    存在对应的产生式并反向压栈    |
|  #AT+   | +i*i#  | 栈顶符号=当前输入符号，指针前移 |
|  #AT    |  i*i#  |    存在对应的产生式并反向压栈    |
|  #ABF   |  i*i#  |    存在对应的产生式并反向压栈    |
|  #ABi   |  i*i#  | 栈顶符号=当前输入符号，指针前移 |
|  #AB    |  *i#   |    存在对应的产生式并反向压栈    |
| #ABF*   |  *i#   | 栈顶符号=当前输入符号，指针前移 |
|  #ABF   |   i#   |    存在对应的产生式并反向压栈    |
|  #ABi   |   i#   | 栈顶符号=当前输入符号，指针前移 |
|  #AB    |   #    |             ε,弹栈              |
|   #A    |   #    |             ε,弹栈              |
|   #     |   #    |            分析成功             |
+---------+---------+----------------------------------+
```

(复制粘贴格式有点跑偏，下图为 prettytable 来进行文法可视化的最终呈现样式)

五、实验结果

terminal 输出结果(由于太长不方便截取图片，直接将 terminal 中的输出拷贝过来了)

```
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3 /Users/aczy156/
学习资料/大二下/编译原理/作业/1 第一次实验/python_ex1/MultiAnalysis/LL1_analysis.py



--------- 在消除左递归之前的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |   E+T    | E->E+T |
|  2   |    E     |    T     |  E->T  |
|  3   |    T     |   T*F    | T->T*F |
|  4   |    T     |    F     |  T->F  |
|  5   |    F     |   (E)    | F->(E) |
|  6   |    F     |    i     |  F->i  |
+------+----------+----------+--------+



--------- 消除左递归之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
|  8   |    B     |    ε     |  B->ε  |
+------+----------+----------+--------+



--------- 提取公因子之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
```

```
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
|  8   |    B     |    ε     |  B->ε  |
+------+----------+----------+--------+


--------- 消除文法左递归的文法的非终结符为 ----------
 ['E', 'T', 'F', 'A', 'B']

--------- 消除文法左递归的文法的终结符为 ----------
 ['(', ')', 'i', '+', '*', '#']


--------- 文法的 FIRST 集为 ----------
FIRST(E) = {(,i}
FIRST(T) = {(,i}
FIRST(F) = {(,i}
FIRST(A) = {ε,+}
FIRST(B) = {*,ε}


--------- 文法的 FOLLOW 集为 ----------
FOLLOW(E) = {#,)}
FOLLOW(T) = {#,+,)}
FOLLOW(F) = {#,*,+,)}
FOLLOW(A) = {#,)}
FOLLOW(B) = {#,+,)}


--------- 该文法对应的预测分析表为 ----------
 +----------+------+------+------+------+------+------+
| 非终结符 |  (   |  )   |  i   |  +   |  *   |  #   |
+----------+------+------+------+------+------+------+
|    E     |  TA  | None |  TA  | None | None | None |
|    T     |  FB  | None |  FB  | None | None | None |
|    F     |  i   | None |  i   | None | None | None |
|    A     | None |  ε   | None | +TA  | None |  ε   |
|    B     | None |  ε   | None |  ε   | *FB  |  ε   |
```

```
+----------+------+------+------+------+------+------+
请输入字符串(exit 跳出循环):i+i*i
分析成功
+--------+--------+--------------------------------+
| 分析栈 | 输入串 |              操作              |
+--------+--------+--------------------------------+
|   #E   | i+i*i# |    存在对应的产生式并反向压栈    |
|  #AT   | i+i*i# |    存在对应的产生式并反向压栈    |
|  #ABF  | i+i*i# |    存在对应的产生式并反向压栈    |
|  #ABi  | i+i*i# | 栈顶符号=当前输入符号，指针前移 |
|  #AB   | +i*i#  |             ε,弹栈             |
|   #A   | +i*i#  |    存在对应的产生式并反向压栈    |
|  #AT+  | +i*i#  | 栈顶符号=当前输入符号，指针前移 |
|  #AT   |  i*i#  |    存在对应的产生式并反向压栈    |
|  #ABF  |  i*i#  |    存在对应的产生式并反向压栈    |
|  #ABi  |  i*i#  | 栈顶符号=当前输入符号，指针前移 |
|  #AB   |  *i#   |    存在对应的产生式并反向压栈    |
| #ABF*  |  *i#   | 栈顶符号=当前输入符号，指针前移 |
|  #ABF  |   i#   |    存在对应的产生式并反向压栈    |
|  #ABi  |   i#   | 栈顶符号=当前输入符号，指针前移 |
|  #AB   |   #    |             ε,弹栈             |
|   #A   |   #    |             ε,弹栈             |
|   #    |   #    |            分析成功            |
+--------+--------+--------------------------------+
请输入字符串(exit 跳出循环):i+i*
分析失败
+--------+--------+--------------------------------+
| 分析栈 | 输入串 |              操作              |
+--------+--------+--------------------------------+
|   #E   | i+i*#  |    存在对应的产生式并反向压栈    |
|  #AT   | i+i*#  |    存在对应的产生式并反向压栈    |
|  #ABF  | i+i*#  |    存在对应的产生式并反向压栈    |
|  #ABi  | i+i*#  | 栈顶符号=当前输入符号，指针前移 |
|  #AB   |  +i*#  |             ε,弹栈             |
|   #A   |  +i*#  |    存在对应的产生式并反向压栈    |
|  #AT+  |  +i*#  | 栈顶符号=当前输入符号，指针前移 |
|  #AT   |  i*#   |    存在对应的产生式并反向压栈    |
|  #ABF  |  i*#   |    存在对应的产生式并反向压栈    |
|  #ABi  |  i*#   | 栈顶符号=当前输入符号，指针前移 |
|  #AB   |   *#   |    存在对应的产生式并反向压栈    |
| #ABF*  |   *#   | 栈顶符号=当前输入符号，指针前移 |
+--------+--------+--------------------------------+
请输入字符串(exit 跳出循环):exit
```

```
Process finished with exit code 0
```

六、实验结论

对实验数据和结果进行分析描述:

在第四步对照实验原理结合每一步已经详细分析。

结论:

在预测分析法设计与实验中，通过代码加深了 LL1 文法的整个过程，消除左递归可以防止分析程序进入死循环，基于非终结符的 FIRST 集和 FOLLOW 集的预测分析表在整个分析过程中起到了分析指引的作用，是整个分析过程的关键。

Code:

在上面已经展示的非常详细，此处不再重复放置。
(此处可以跳转至 GitHub 仓库查看
https://github.com/Aczy156/Compiling-Principle-Work)

七、实验小结

在本次实验中，是大学时代第一次课程作业用 python 完成，三个实验的总代码量在八九百行左右，收获颇丰。在原来其他学科的课设中，熟悉了 Java 和 C/C++，跳出舒适圈用 python 之后，深刻体会到了 python 的强大的一些已有的数据结构，例如字典，原来在一些小项目中已经用过了，这次更加深了印象；以及强大的字符串处理功能。

然后在 coding 的过程中，因为自上而下的这两种方法的学习时间较久，然后又跟着陈老师的视频重新温习了一边，才得以写出来。

在一些子模块中，就拿消除左递归来说，最一开始并没有按照老师讲的双层循环实现，而是通过三层循环来遍历最后暴力出来的，虽然也可以消除左递归，但是这样并没有按照老师的思路去走，于是最终还是通过查资料，学习其他人的思想来修改了那个模块；再如 FIRST 集和 FOLLOW 集的获取，最一开始并没有注意到多次扫描，导致一些符号总是漏下，因为只要有一个改变就要重新扫描，后来通过增加扫描次数才得以更正。

总之，在此次的实验中，不仅收获了 python 编程语言的代码能力，更体会到了使用 python 一行顶十行的乐趣，体会到了自己 coding 的 code 可以 compile 自己写的 C 的快乐，与此同时，更加深了自上而下中预测分析法的知识结构。

《编译原理》

实验报告

项目名称 实验三《递归下降分析法设计与实现》

专业班级 软件 1805 班

学　　号 8209180516

姓　　名 陈冉飞

实验成绩：

批阅教师：

2020 年 5 月 22 日

实验 3《递归下降分析法设计与实现》

实验学时：_____ 实验地点：_____ 实验日期：_____

一、实验目的

根据某一文法编制调试递归下降分析程序，以便对任意输入的符号串进行分析。本次实验的目的主要是加深对递归下降分析法的理解。

二、实验内容

程序输入/输出示例（以下仅供参考）：

对下列文法，用递归下降分析法对任意输入的符号串进行分析：

（1）E- TG

（2）G- +TG|—TG

（3）G- ε

（4）T- FS

（5）S- *FS|/FS

（6）S- ε

（7）F- (E)

（8）F- i

输出的格式如下：

递归下降分析程序，编制人：姓名，学号，班级

输入一以#结束的符号串(包括+—*/（）i#)：在此位置输入符号串例如：i+i*i# (3)输出结果：i+i*i#为合法符号串

备注：输入一符号串如 i+i*#,要求输出为“非法的符号串”。

注意：

表达式中允许使用运算符（+-*/）、分割符（括号）、字符 I，结束符#；

如果遇到错误的表达式，应输出错误提示信息（该信息越详细越好）。

三、实验方法

实验二、实验三、实验四在一个项目档案中，均是采用 python 进行项目开发，以 pycharm 作为 IDE，并且利用 GitHub(https://github.com/Aczy156/Compiling-Principle-Work)进行项目管理，对项目结构进行模块化管理，将一些常用到的

模块提取出来，方便不同的分析方法公用

四、实验步骤

(一)整体结构(如右图，该程序总代码量较少，但是其中用到的消除左递归和提取公因子在 tools 下在实验二中已经有体现)

递归下降子程序是递归下降分析的关键，在构思这个实验的时候，通过在每次进入下一层递归之间做好判断，然后返回这一层之后也做好判断，从而通过一个程序来模拟所有的非终结符的与之对应的递归下降子程序(就不用针对每一个非终结符再去写该终结符对应的子程序)。

(二)详细过程

①读入文法并做文法预处理

```
""" 文法字典预处理 """
for i in self.grammer:
    self.grammer[i] = str(self.grammer[i][0]).split('|')
# print('==========', self.grammer, '+++++++', self.vn)
draw_grammer.draw_grammer(grammer=self.grammer, vn=self.vn, descrpition='原始文法')  # 打印文法
```

②消除左递归：(和实验二一样，利用封装好的 tools 中的 Eliminate_Left_Recursion.py)

```
""" 消除左递归 """
eliminate_left_recursion = EliminateLeftRecursion(grammer=self.grammer, vn=self.vn)
self.grammer, self.vn = eliminate_left_recursion.remove_left_recursion()
# print('==========', self.grammer, '+++++++', self.vn)
draw_grammer.draw_grammer(grammer=self.grammer, vn=self.vn, descrpition='消除左递归之后的文法')  # 打印文法
```

③提取公因子：(和实验二一样，利用封装好的 tools 中的 Extract_Common_Factors.py)

```
""" 提取公因子 """
extractcommonfactors = ExtractCommonFactors(grammer=self.grammer, vn=self.vn)
self.grammer, self.vn = extractcommonfactors.remove_common_factor()
# print('==========', self.grammer, '+++++++', self.vn)
draw_grammer.draw_grammer(grammer=self.grammer, vn=self.vn, descrpition='提取公因子之后的文法')  # 打印文法
```

④进行递归下降子程序分析

```
""" 递归下降分析 """
while True:
    self.goal_str = input('请输入字符串(exit 跳出循环):')
    self.p = 0
    if self.goal_str == 'exit':
        break
    flag = (self.match(self.vn[0]) & (self.p == len(self.goal_str)))  # 必须要遍历完测试字符串
    if flag:
        print(self.goal_str + ' 分析成功')
    else:
        print(self.goal_str + ' 分析失败')
```

如果碰到非终结符，那么就直接递归这个非终结符的子程序，往下层去 match。

```
def match(self, ch):
    """ 递归下降进行匹配，通过模拟所有的非终结符的子程序模块 """
    for i in range(len(self.grammer[ch])):
        rule = self.grammer[ch][i]
        # print(i, '--->>>', rule)
        record_p = self.p  # 记录指针位置，方便回溯
        flag = True
        for item in rule:
            if item in self.vn:
                flag = self.match(item)  # 如果碰到了非终结符，直接递归非终结符的子程序
                if flag == 0: break
            elif self.p < len(self.goal_str) and item == self.goal_str[self.p]:
                self.p += 1
            elif item == 'ε':  # 如果 rule 中所有非ε都已经遍历，就遍历ε
                break
            else:
                flag = False
                break
        if flag == 0:
            self.p = record_p
            continue
        else:
            return True
    return False
```

⑤⑥⑦⑧⑨⑩

五、实验结果

```
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3 /Users/aczy156/学习资料/大二下/编译原理/作业/1 第一次实验/python_ex1/MultiAnalysis/RecursiveDescent_analysis.py


--------- 原始文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |   E+T    | E->E+T |
|  2   |    E     |    T     |  E->T  |
|  3   |    T     |   T*F    | T->T*F |
|  4   |    T     |    F     |  T->F  |
|  5   |    F     |   (E)    | F->(E) |
|  6   |    F     |    i     |  F->i  |
+------+----------+----------+--------+


--------- 消除左递归之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
|  8   |    B     |    ε     |  B->ε  |
+------+----------+----------+--------+


--------- 提取公因子之后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    E     |    TA    | E->TA  |
|  2   |    T     |    FB    | T->FB  |
|  3   |    F     |   (E)    | F->(E) |
|  4   |    F     |    i     |  F->i  |
|  5   |    A     |   +TA    | A->+TA |
|  6   |    A     |    ε     |  A->ε  |
|  7   |    B     |   *FB    | B->*FB |
```

```
|  8   |    B     |    ε     | B->ε  |
+------+----------+----------+-------+
请输入字符串(exit 跳出循环):i+i*i
i+i*i 分析成功
请输入字符串(exit 跳出循环):i*
i* 分析失败
请输入字符串(exit 跳出循环):i+(i+i*(i+i))
i+(i+i*(i+i)) 分析成功
请输入字符串(exit 跳出循环):(i)
(i) 分析成功
请输入字符串(exit 跳出循环):(i+i*i)
(i+i*i) 分析成功
请输入字符串(exit 跳出循环):i+(i+i*(i+i)))
i+(i+i*(i+i))) 分析失败
请输入字符串(exit 跳出循环):exit

Process finished with exit code 0
```

六、实验结论

对实验数据和结果进行分析描述:

在第四步对照实验原理结合每一步已经详细分析。

结论:

通过模拟所有非终结符的递归下降子程序的实现过程, 体会到递归下降子程序在分析过程中起到重要作用，不同的非终结符对应的递归下降子程序是不同的(因为要依据产生式去考虑)，才能做到递归分析。

Code：(https://github.com/Aczy156/Compiling-Principle-Work**)**

在代码实现的过程中主要体现在了递归不同子程序的过程, 在递归的每一层都要搞清楚当前的这一层是哪个非终结符, 然后无论是进入这一层递归还是出去这一层递归都要结合着产生式来看, 看这个非终结符的左右的情况, 来做该非终结符对应的进入和返回的特定操作。

七、实验小结

在本次实验中，基于上一次实验也就是实验二的基础，完成此次实验相对较于轻松，尤其是对于递归程序，写法很像 DFS 深搜，写习惯之后对于每层之间的递归都相对更加了解，而且类似于 DFS 深搜也需要在对每一层进行特殊的讨论，这也就对应这每一个非终结符的不同的递归子程序。

在本次实验中也碰到了一些问题，例如一开始就 coding 递归的代码，结果没有进行消除左递归等预处理，然后递归过程的一些判断条件也是如果不添加会导致无法跳出，需要注意。

《编译原理》

实验报告

项目名称 实验四《算符优先分析法设计与实现》

专业班级 软件 1805 班

学　　号 8209180516

姓　　名 陈冉飞

实验成绩：

批阅教师：

2020 年 5 月 22 日

实验 4《算符优先分析法设计与实现》

实验学时：_____ 实验地点：_____ 实验日期：_____

一、实验目的

加深对语法分析器工作过程的理解；加强对算符优先分析法实现语法分析程序的掌握；能够采用一种编程语言实现简单的语法分析程序；能够使用自己编写的分析程序对简单的程序段进行语法翻译。

二、实验内容

用算符优先分析法编制语法分析程序，语法分析程序的实现可以采用任何一种编程语言和工具。

三、实验方法

利用 python 语言，IDE 使用 pycharm，利用 GitHub(https://github.com/Aczy156/Compiling-Principle-Work)进行项目管理。。

四、实验步骤

(一)整体结构

整体利用 FIRSTVT 来判断<·，然后通过 LASTVT 来判断·>，然后通过文法规则来判断·=，并填充到算符优先表中，然后利用算符分析表来进行分析，在分析的过程中利用非语法树(也就是最左素短语的左右两边的算符都大于相邻的，中间凸起形状)从而实现，进而而提取最左素短语，最后检查最后分析的结果来判断是否成功分析。

(二)详细过程

①文法输入以及预处理

```
def grammar_preprocess(g):
    """ 文法预处理 """
    result = defaultdict(set)
    first_left = ''
    for line in re.split('\n', g):
        line = re.compile(r'\s+').sub('', line)
        m = re.compile(r'([A-Z])->(.+)').match(line)
        if not m:
            break
        # 获取左部和右部
        left = m.group(1)
        if not first_left:
            first_left = left
        right = m.group(2)
        # 判断是否为算符文法
        if re.compile(r'[A-Z]{2,}').search(right):
            return False
        # 切割右部并将右部添加到结果集
        right = right.split('|')
```

```
        result[left] |= set(right)

    """ 文法归一化 """
    propressed_table = PrettyTable(['编号', '箭头左边', '箭头右边', '产生式'])
    propressed_table.add_row([1, 'S', 'E', 'S->E'])
    idx = 2
    for l, r in result.items():
        for p in r:
            propressed_table.add_row([idx, l, p, l + '->' + p])
            idx += 1
    print('\n---------- 经过归一化后的文法 ----------\n', propressed_table)
    result['S'] = {'#{0}#'.format(first_left)}
    return [(l, p) for l, r in result.items() for p in r]
```

②获取 FIRSTVT 集和 LAST 集

获取 FIRSTVT 集

1、若有产生式 P→a…或 P→Qa…，则 a∈FIRSTVT(P)

2、若 a∈FIRSTVT(Q)，且有产生式 P→Q…，则 a∈FIRSTVT(P)

Code FIRSTVT 集的获取，通过利用多次扫描来将无限空间缩小为有限空间从而来覆盖所有的 FIRSTVT 集和获取，在多次扫描的过程中通过一个 updated 来标记看看什么时候没有任何非终结符的 FIRSTVT 改变，但是只要有一个改变，就要重新扫描一边，防止这个改变的影响到其他的。

```
def compute_firstvt(g):
    """输入一个算符文法的语法规则，输出所有非终结符的 FIRSTVT 集合"""
    result = defaultdict(set)

    # 添加由文法规则本身得到的集合
    for left, right in g:
        new_set = result[left]
        if len(right) >= 1 and not right[0].isupper():
            new_set.add(right[0])
        elif len(right) >= 2:
            new_set.add(right[1])

    # 迭代添加其他集合
    updated = True
    while updated:
        updated = False
        for left, right in g:
            new_set = set(result[left])
            if len(right) >= 1 and right[0].isupper():
                new_set |= result[right[0]]
```

```
            if len(new_set) > len(result[left]):
                result[left] = new_set
                updated = True

    return dict(result)
```

获取 LASTVT 集

1、若有产生式 P→⋯ a 或 P→ ⋯ aQ，则 a∈LASTVT(P)

2、若 a∈LASTVT(Q)，且有产生式 P→⋯ Q，则 a∈LASTVT(P)

Code LASTVT 集的获取，通过利用多次扫描来将无限空间缩小为有限空间从而来覆盖所有的 LASTVT 集和获取, 在多次扫描的过程中通过一个 updated 来标记看看什么时候没有任何非终结符的 LASTVT 改变，但是只要有一个改变，就要重新扫描一边，防止这个改变的影响到其他的。

```
def compute_lastvt(g):
    """输入一个算符文法的语法规则，输出所有非终结符的 LASTVT 集合"""

    result = defaultdict(set)

    # 添加由文法规则本身得到的集合
    for left, right in g:
        new_set = result[left]
        if len(right) >= 1 and not right[-1].isupper():
            new_set.add(right[-1])
        elif len(right) >= 2:
            new_set.add(right[-2])

    # 迭代添加其他集合
    updated = True
    while updated:
        updated = False
        for left, right in g:
            new_set = set(result[left])
            if len(right) >= 1 and right[-1].isupper():
                new_set |= result[right[-1]]
            if len(new_set) > len(result[left]):
                result[left] = new_set
                updated = True

    return dict(result)
```

③根据 FIRSTVT 集和 LASTVT 集来构造算符分析表

通过检查 G 的每个产生式的每个候选式，可找出所有满足 ab 的终结符对。

根据 FIRSTVT 和 LASTVT 集合，检查每个产生式的候选式，确定满足关系<·和·>的所有终结符对

假定有个产生式的一个候选形为 …aP…， 那么，对任何 b∈FIRSTVT(P)，有 a<·b

假定有个产生式的一个候选形为…Pb…， 那么，对任何 a∈LASTVT(P)，有 a·>b

```
def compute_prior(grammar, firstvt, lastvt):
    """ 算符文法的语法规则、非终结符的FIRSTVT、LASTVT 集合 填入算符优先分析表,按照·= ·>
<· 的顺序"""
    vn = {c for left, right in grammar
          for c in right
          if not c.isupper()}
    result = {k: {k: '  ' for k in vn} for k in vn}

    for left, right in grammar:  # 利用 grammer 即 i 和 i+2 位置上的来计算 '·='
        for i in range(len(right) - 1):
            if right[i].isupper():
                pass
            elif not right[i + 1].isupper():
                result[right[i]][right[i + 1]] = '·='
            elif i + 2 < len(right):
                result[right[i]][right[i + 2]] = '·='

    for left, right in grammar:  # 利用 firstvt 来计算 '<·'
        for i in range(len(right) - 1):
            if right[i].isupper() or not right[i + 1].isupper():
                continue
            a = right[i]
            for b in firstvt[right[i + 1]]:
                if result[a][b] != '<·' and result[a][b] != '  ':
                    return False
                result[a][b] = '<·'

    for left, right in grammar:  # 利用 lastvt 来计算 '<·'
        for i in range(len(right) - 1):
            if not right[i].isupper():
                continue
            b = right[i + 1]
            for a in lastvt[right[i]]:
                if result[a][b] != '·>' and result[a][b] != '  ':
                    return False
                result[a][b] = '·>'
```

```
    # print(result)
    return result
```

④寻找最左素短语#N1a1N2a2…aj-1**Njaj…NiaiNi+1**ai+1...NnanNn+1#进行算符优先分析，知道最后完成分析，如果不是算符文法，在返回的第一个元素中通过 True or False 来进行区分，然后如果是，通过解析 result 来展示整个过程。

```
def solve(grammar, prior_table, sentence):
    """ 算法有限分析实现：结合算符有限表prior_table 和文法grammer 来检测句子
goal_str"""

    def update_states(states, new_states):
        for state, seq in new_states.items():
            if state not in states:
                pass
            elif len(seq) < len(states[state]):
                pass
            else:
                continue
            states[state] = seq

    def reduce_single(grammar, states):
        found_states = list(states.keys())
        for state in found_states:
            lastnt = state[-1]
            for left, right in grammar:
                if right != lastnt:
                    continue
                new_state = state[:-1] + left
                if new_state in found_states:
                    continue
                states[new_state] = states[state] + \
                                    [(state, '{0}->{1}'.format(left, right))]
                found_states.append(new_state)

    def reduce_(grammar, states, c):
        new_states = {}
        for state, seq in states.items():
            add_to_new = False

            if len(state) < 1:
                add_to_new = True
            elif not state[-1].isupper():
                if state[-1] not in prior_table:
                    return {}
```

```
            if prior_table[state[-1]][c] in ['<·', '·=']:
                add_to_new = True
        elif len(state) < 2:
            add_to_new = True
        elif state[-2] not in prior_table:
            return {}
        elif prior_table[state[-2]][c] in ['<·', '·=']:
            add_to_new = True

        # 如果不需要归约直接添加
        if add_to_new:
            update_states(new_states, {
                state + c: seq + [(state, None)]})
            continue

        # 进行归约
        for left, right in grammar:
            if not state.endswith(right):
                continue
            states_to_reduce = {
                state[:-len(right)] + left:
                    seq + [(state, '{0}->{1}'.format(left, right))]
            }
            reduce_single(grammar, states_to_reduce)
            update_states(new_states,
                          reduce_(grammar, states_to_reduce, c))

    return new_states

states = {'': []}
sentence = '#{0}#'.format(sentence)
for i, c in enumerate(sentence):
    states = reduce_(grammar, states, c)
    if not states:
        return False, sentence[:i]

# 寻找最短归约路径
result = None
for seq in states.values():
    if not result or len(seq) < len(result):
        # print('seq == ', seq)
        result = seq

return True, result
```

五、实验结果

```
/Library/Frameworks/Python.framework/Versions/3.7/bin/python3 /Users/aczy156/学习资料/大二下/编译原理/作业/1 第一次实验/python_ex1/MultiAnalysis/OperatorFirst_analysis.py

--------- 经过归一化后的文法 ----------
 +------+----------+----------+--------+
| 编号 | 箭头左边 | 箭头右边 | 产生式 |
+------+----------+----------+--------+
|  1   |    S     |    E     |  S->E  |
|  2   |    E     |    T     |  E->T  |
|  3   |    E     |   E+T    | E->E+T |
|  4   |    T     |    F     |  T->F  |
|  5   |    T     |   T*F    | T->T*F |
|  6   |    F     |    P     |  F->P  |
|  7   |    F     |   P^F    | F->P^F |
|  8   |    P     |   (E)    | P->(E) |
|  9   |    P     |    i     |  P->i  |
+------+----------+----------+--------+

--------- 各个非终结符的 FIRSTVT 集为 ----------

FIRSTVT(P) = {'(', 'i'}
FIRSTVT(F) = {'(', '^', 'i'}
FIRSTVT(S) = {'#'}
FIRSTVT(E) = {'*', '+', '(', 'i', '^'}
FIRSTVT(T) = {'*', '(', 'i', '^'}

--------- 各个非终结符的 LASTVT 集为 ----------

LASTVT(P) = {')', 'i'}
LASTVT(F) = {')', 'i', '^'}
LASTVT(S) = {'#'}
LASTVT(E) = {'*', '+', 'i', '^', ')'}
LASTVT(T) = {'*', 'i', '^', ')'}

--------- 该文法的算符有限分析表为 ----------

   | #  | +  | (  | i  | ^  | )  | *
-------------------------------------
 # | ·= | <· | <· | <· | <· |    | <·
-------------------------------------
```

```
 + | ·> | ·> | <· | <· | <· | ·> | <·
-------------------------------------
 ( |    | <· | <· | <· | <· | ·= | <·
-------------------------------------
 i | ·> | ·> |    |    | ·> | ·> | ·>
-------------------------------------
 ^ | ·> | ·> | <· | <· | <· | ·> | ·>
-------------------------------------
 ) | ·> | ·> |    |    | ·> | ·> | ·>
-------------------------------------
 * | ·> | ·> | <· | <· | <· | ·> | ·>
请输入字符串(exit 跳出循环):i+i*i
分析成功
+------+--------+--------+-------------+
| 编号 |   栈   | 符号串 |     动作    |
+------+--------+--------+-------------+
|  0   |   #    | i+i*i# |    移进     |
|  1   |   #i   | +i*i#  |    移进     |
|  2   |   #P   | +i*i#  |  规归 P->i  |
|  3   |   #F   | +i*i#  |  规归 F->P  |
|  4   |   #T   | +i*i#  |  规归 T->F  |
|  5   |   #E   | +i*i#  |  规归 E->T  |
|  6   |  #E+   |  i*i#  |    移进     |
|  7   |  #E+i  |  *i#   |    移进     |
|  8   |  #E+P  |  *i#   |  规归 P->i  |
|  9   |  #E+F  |  *i#   |  规归 F->P  |
|  10  |  #E+T  |  *i#   |  规归 T->F  |
|  11  | #E+T*  |   i#   |    移进     |
|  12  | #E+T*i |   #    |    移进     |
|  13  | #E+T*P |   #    |  规归 P->i  |
|  14  | #E+T*F |   #    |  规归 F->P  |
|  15  |  #E+T  |   #    | 规归 T->T*F |
|  16  |   #E   |   #    | 规归 E->E+T |
+------+--------+--------+-------------+
请输入字符串(exit 跳出循环):i*(i)
分析成功
+------+--------+--------+-------------+
| 编号 |   栈   | 符号串 |     动作    |
+------+--------+--------+-------------+
|  0   |   #    | i*(i)# |    移进     |
|  1   |   #i   | *(i)#  |    移进     |
|  2   |   #P   | *(i)#  |  规归 P->i  |
|  3   |   #F   | *(i)#  |  规归 F->P  |
|  4   |   #T   | *(i)#  |  规归 T->F  |
```

```
|  5   |  #T*   |  (i)#  |    移进     |
|  6   |  #T*(  |  i)#   |    移进     |
|  7   | #T*(i  |   )#   |    移进     |
|  8   | #T*(P  |   )#   |  规归 P->i  |
|  9   | #T*(F  |   )#   |  规归 F->P  |
|  10  | #T*(T  |   )#   |  规归 T->F  |
|  11  | #T*(E  |   )#   |  规归 E->T  |
|  12  | #T*(E) |   #    |    移进     |
|  13  |  #T*P  |   #    | 规归 P->(E) |
|  14  |  #T*F  |   #    |  规归 F->P  |
|  15  |   #T   |   #    | 规归 T->T*F |
+------+--------+--------+-------------+
请输入字符串(exit 跳出循环):i(i)
分析失败
请输入字符串(exit 跳出循环):i*(i^i+i*i)
分析成功
+------+-----------+-------------+-------------+
| 编号 |    栈     |   符号串    |    动作     |
+------+-----------+-------------+-------------+
|  0   |     #     | i*(i^i+i*i)# |    移进     |
|  1   |    #i     | *(i^i+i*i)#  |    移进     |
|  2   |    #P     | *(i^i+i*i)#  |  规归 P->i  |
|  3   |    #F     | *(i^i+i*i)#  |  规归 F->P  |
|  4   |    #T     | *(i^i+i*i)#  |  规归 T->F  |
|  5   |   #T*     |  (i^i+i*i)#  |    移进     |
|  6   |   #T*(    |  i^i+i*i)#   |    移进     |
|  7   |  #T*(i    |   ^i+i*i)#   |    移进     |
|  8   |  #T*(P    |   ^i+i*i)#   |  规归 P->i  |
|  9   |  #T*(P^   |   i+i*i)#    |    移进     |
|  10  | #T*(P^i   |    +i*i)#    |    移进     |
|  11  | #T*(P^P   |    +i*i)#    |  规归 P->i  |
|  12  | #T*(P^F   |    +i*i)#    |  规归 F->P  |
|  13  |  #T*(F    |    +i*i)#    | 规归 F->P^F |
|  14  |  #T*(T    |    +i*i)#    |  规归 T->F  |
|  15  |  #T*(E    |    +i*i)#    |  规归 E->T  |
|  16  |  #T*(E+   |    i*i)#     |    移进     |
|  17  | #T*(E+i   |     *i)#     |    移进     |
|  18  | #T*(E+P   |     *i)#     |  规归 P->i  |
|  19  | #T*(E+F   |     *i)#     |  规归 F->P  |
|  20  | #T*(E+T   |     *i)#     |  规归 T->F  |
|  21  | #T*(E+T*  |     i)#      |    移进     |
|  22  | #T*(E+T*i |      )#      |    移进     |
|  23  | #T*(E+T*P |      )#      |  规归 P->i  |
|  24  | #T*(E+T*F |      )#      |  规归 F->P  |
```

```
|  25  |  #T*(E+T  |      )#      | 规归 T->T*F  |
|  26  |   #T*(E   |      )#      | 规归 E->E+T  |
|  27  |   #T*(E)  |      #       |    移进      |
|  28  |    #T*P   |      #       | 规归 P->(E)  |
|  29  |    #T*F   |      #       |  规归 F->P   |
|  30  |     #T    |      #       | 规归 T->T*F  |
+------+-----------+--------------+--------------+
请输入字符串(exit 跳出循环):exit

Process finished with exit code 0
```

六、实验结论

对实验数据和结果进行分析描述:

在第四步对照实验原理结合每一步已经详细分析。

结论:

利用自上而下的分析法通过代码实现，可以理解到从输入串开始，逐步归约，直到文法的开始符号的整体过程，在过程中通过归约和文法的产生式规则，把串中出现的产生式的右部替换成左部符号按照算符的优先关系和结合性质进行语法分析。(个人理解：结合上面的 LL1 文法来说，算符优先文法更适合分析表达式)

Code：(https://github.com/Aczy156/Compiling-Principle-Work**)**

在实验中对于 FIRSTVT 集和 LASTVT 集的获取，都是通过利用多次扫描来将无限空间缩小为有限空间从而来覆盖所有的 FIRSTVT 集和 LASTVT 集的获取，类似与前面 LL1 文法的 FIRST 集和 FOLLOW 集的获取，对实验数据和结果进行分析描述，给出实验取得的成果和结论。

七、实验小结

在本次实验中，难度较为上一个递归下降分析相对有一定提升，然后思路和实验二实验三也相差较多，通过模拟整个过程，理解了算符有限分析的算法思想。

通过 python 编程，提升了 python 语言的 coding 和 debug 能力，并对 python 库函数中 re 字符串处理、collections 容器数据管理更加熟悉。

三个实验整体的感受:

通过这三个实验放在一起整体开发，对比了算法之间的不同点与相同点，(个人理解)，认为类似算法都可以归结为一个**算法实现的理论知识和总控程序分离的框架**，总控程序可以利用输入串、理论知识得到的表、各式各样的集合等等来对这些子模块进行调度控制，然后理论知识的核心体现在预测分析表、算符有限表等，以及各种辅助的集合等等，但是大致都可以提炼出来控制+理论知识模块来完成分析。

在本次实验中，虽然花费时间较长但是收获了编译原理知识和 python coding 能力提升的双丰收：一些算法的知识已经不太熟悉，因为过去的时间比较久了，然后还需要重新花费两到三天的时间来把陈老师的视频重新看一遍，然后再进行编码，在编码过程中遇到困难通过搜索、查阅相关资料来克服困难，现在熟悉每一个算法的核心思想，收获蛮多，进步很大；python coding 能力也得到提升，原来虽然用 python 写项目，但是还并没有达到这个代码量，

在这次中通过使用各种语法，提升了 coding 能力，总而言之，这次实验是一个非常有意义的实验!