深 圳 大 学 实 验 报 告

课程名称：编译原理

实验项目名称：自顶向下的语法分析程序设计

学院：计算机与软件学院

专业：软件工程

指导教师：蔡树彬

报告人：郑彦薇 学号：2020151022 班级：软件工程 01 班

实验时间：2023 年 4 月 18 日至 5 月 16 日

实验报告提交时间：2022 年 5 月 24 日

教务处制

实验目的与要求：

目的：通过设计、实现基于递归下降或预测分析法的语法分析程序，掌握并能应用自顶向下语法分析技术进行语法分析。

要求:

第一部分：First 集和 Follow 集合的构造

给定一个文法，如

简单算术表达式：E→E+T | T ...

简单布尔表达式：B→B and U | U ...

赋值语句：A→ V = E | B

条件语句：C→ if (B) L else L

语句：S→ A | C

语句块：L→ S;L | S

程序：P→L

如何将改造成 LL（1）文法，再求出各个非终结符的 First 和 Follow 集合。

第二部分：LL（1）分析表的构造和 LL（1）预测分析法的分析过程

对给定 LL（1）文法，构造其 LL（1）分析表。然后对给定的若干字符串，给出字符串的预测分析过程。

第三部分：语法树的构造与输出

对第二部分的预测分析过程，构造对应的语法树并输出。

方法、步骤：

要完成本实验，依据实验要求进行分解，需要完成的实验步骤是：

1. 你是如果改造原文法，使其称为 LL（1）文法的？

答：可以通过 3 个步骤实现：①消除左递归。对于一个形如 A→Aα|β的产生式，可以将其改写为 A→βA’、A’→αA’|ε。②提取左公因子。对于一组具有公共前缀的产生式，可以将其提取为一个新的非终结符，并将其作为原终结符的替代。如对于 S→ab|ac|ad，可以变为 S→aS’，S’→b|c|d。③为每个非终结符求出 First 和 Follow 集合，并检查是否有 First 集和 Follow 集交集不为空的情况。如果有，则对文法进行进一步的改造（左因子分解、引入新的非终结符、改变产生式顺序等）。

2. 你是如何求各个非终结符的 First 集合的？

答：对于一个非终结符 A，可以通过以下步骤求出其 First 集：①如果 A 是一个终结符，那么它的 First 集就是它本身。②对于 A 的所有产生式，如果第一个符号是终结符或者空串，那么这一符号属于 A 的 First 集。③对于 A 的所有产生式，如果第一个符号是非终结符 B，则 B 的 First 集中，除空串外的符号都属于 A 的 First 集；如果 B 的 First 集中包含空串，则继续查看产生式的下一个符号，直到遇到终结符或产生式结束。④如果 A 的所有产生式都包含空串，则将空串加入 A 的 First 集中。⑤重复上述步骤，直到所有非终结符的 First 集都求出来为止。

3. 你是如何求各个非终结符的 Follow 集合的？

答：对于一个非终结符 A，可以通过以下步骤求出其 Follow 集：①将$符号加入到文法起始符号 S 的 Follow 集中。②对于每个产生式 A→αBβ，将 B 的 Follow 集合中所有非空符号加入到β的 First 集合中；如果β可以推导出空串，则将 B 的 Follow 集合中的符号加入到 Follow(A)中。③对于每个产生式 A→αB，将 A 的 Follow 集合加入到 B 的 Follow 集合中。④重复上

述步骤，直到所有非终结符的 Follow 集都求出来为止。

4. 你是如何设计 LL（1）预测分析表，并如何构造出来的？

答：可以按照以下方法设计 LL（1）预测分析表：对于每个非终结符，首先求解其 First 集和 Follow 集。然后对于每个形如 A→α的产生式，将所有的符号串 First(α)中的终结符 a，都加入到 M[A，a]中。如果ε∈First(α)，则将所有 Follow(A)中的终结符 a，都加入到 M[A，a]中。

可以按照以下步骤构造出 LL（1）预测分析表：先初始化分析表 M，将所有 M[A，a]的值都设为空。然后对于每个非终结符 A 和终结符 a，如果 M[A，a]中有多个产生式，则该文法不是 LL（1）文法，否则将产生式 A→α添加到 M[A，a]中。对于每个非终结符 A 和终结符 a，如果 M[A，a]为空，则跳过此项。最后判断对于每个终结符 a，M[S，a]是否为空，如果为空，且其中 S 为文法的起始符号，则该文法不是 LL（1）文法。

5. 你是如何实现 LL（1）预测分析法的？

答：结合上述 LL 预测分析表的构造和 First 集以及 Follow 集的求解，可以通过以下几步实现 LL（1）预测分析法：①判断文法有没有左递归，如果有，则对该文法消除左递归。②对处理后的文法进行遍历，求出非终结符和终结符。③求出每个符号的 First 集和 Follow 集。④判断是不是 LL（1）文法，如果是则构造预测分析表，并可以输入句子模拟输出的进栈出栈情况。

6. 你是如何设计语法树的存储结构，并如何结合 LL（1）预测分析法构成出来？

答：可以使用节点和指针设计语法树的存储结构。每个节点代表一个语法规则的非终结符或终结符，指针用于连接子节点或兄弟节点。在使用 LL（1）预测分析法时，首先构造 First 集和 Follow 集，并根据文法产生式的左部非终结符和右部的 First 集来构造预测分析表，然后利用该表对输入串进行语法分析，同时构建语法树。具体可以通过以下步骤实现：

①定义语法树的节点结构，包括节点类型、节点值、子节点指针和兄弟节点指针等信息。

②构造文法的 First 集和 Follow 集，根据 First 和 Follow 集构造预测分析表。

③初始化语法树的根节点，初始化分析栈和输入栈。

④从输入栈中取出当前输入符号，并从分析栈中取出栈顶符号。

⑤如果栈顶符号是终结符号，则将其与输入符号进行匹配。如果匹配成功，则将输入栈中下一个符号压入分析栈，同时将当前节点作为栈顶节点的子节点。

⑥如果栈顶符号是非终结符号，则根据预测分析表查找对应的产生式。将产生式右部符号依次压入分析栈，同时将当前节点作为栈顶节点的子节点。

⑦重复 4~6，直到分析栈为空或输入栈为空。

⑧如果分析成功，则返回语法树的根节点。否则返回错误信息。

实验过程及内容：

一、First 和 Follow 集合的构造

参照方法、步骤中的 1~3。

1. 对于题目给定的文法，在 python 中利用字典进行存储

```
# 定义文法的产生式
productions = {
    'E': ['E+T', 'T'],
    'T': ['int', 'float', 'double', 'long'],
    'B': ['B and U', 'U'],
    'U': ['true', 'false'],
    'V': ['var'],
    'A': ['V=E', 'B'],
    'C': ['if (B) L else L'],
    'S': ['A', 'C'],
    'L': ['S;L', 'S'],
    'P': ['L']
}
```

根据方法步骤中第一个问题的求解思路，接着需要对文法中每个非终结符的First集和Follow集进行求解。

2. 求解各个非终结符的 First 集

参考方法步骤中的 2。该过程的实现基于迭代，每次迭代遍历所有产生式，检查是否有新的首终结符加入到集合中。如果有，则需要再次进行迭代，否则退出。是否有新的首终结符通过 bool 类型数据 updated 进行判断。

```
first = {}
for key in productions:
    first[key] = set()
while True:
    updated = False
```

在迭代中，遍历每一个产生式。对于每一个产生式，从第一个符号开始。首先判断是否是终结符，如果是，则将其加入到当前符号的 first 集中。

```
if production[0] not in productions.keys():
    # 如果第一个符号是终结符，则将其放入key的first集中
    first[key].add(production[0])
    updated = True
```

对于不是非终结符的，将其对应的 First 集中的元素加入到当前 key 的 first 集中，并继续处理下一个符号，如果所有符号都处理完毕，说明该产生式可以推导出空串，那么将空串加入到 key 的 first 集中。

```
for symbol in production:
    if symbol in first.keys():
        first[key] |= first[symbol]
        if 'ε' not in first[symbol]:
            break
    else:
        first[key].add(symbol)
        break
else:
    first[key].add('ε')
    updated = True
```

最后对 updated 值进行判断，如果没有新的首终结符加入到集合中，则迭代结束。

```
    if not updated:
        break
return first
```

3. 求解各个非终结符的 Follow 集

参照方法步骤中的 3。对于每个非终结符，初始化其 Follow 集为空集，并将起始符号 S 的 Follow 集加上结束符号‘$’。

```
follow = {}
for key in productions:
    follow[key] = set()
follow['P'].add('$')
```

设置迭代，不断执行以下循环体直到不再有新的符号加入 Follow 集。这里同样可以设置一个 updated 值判断是否有新的符号加入 Follow 集。在循环体中：

对于每个非终结符的每个产生式，遍历其中的符号。判断是不是非终结符并判断当前符号是否是产生式的最后一个符号，如果是则将其 Follow 集加上该产生式中该符号后面的所有非空 First 集。

```
# 遍历产生式中的每个符号
for i, symbol in enumerate(production):
    # 如果当前符号是一个非终结符号
    if symbol in productions.keys():
        if i == len(production) - 1:  # 最后一个符号
            follow[symbol] |= follow[key]
```

如果不是最后一个符号，则继续遍历当前符号后面的所有符号 j。如果对应符号 j 的 First 集非空，则将该非终结符的 Follow 集加上该符号的 First 集。或判断出该符号的 First 集不包含空串，跳出循环。如果判断的该符号是终结符号，则将其加入该非终结符的 Follow 集。

```
for j in range(i+1, len(production)):
    if production[j] in first.keys():
        follow[symbol] |= first[production[j]]
        # 如果该符号的First集不包含空串，则跳出循环
        if 'ε' not in first[production[j]]:
            break
    else:
        # 如果该符号是终结符号，则将其加入该非终结符的Follow集
        follow[symbol].add(production[j])
        break
```

如果符号后面的所有符号的 First 集都包含空串，则该非终结符的 Follow 集需要加上其所在产生式所在非终结符的 Follow 集。

```
# 如果符号后面的所有符号的First集都包含空串，则将该非终结符的Follow集加上该非终结符所在产生式所在非终结符的Follow集
follow[symbol] |= follow[key]
```

最后对 updated 值进行判断，如果没有新的符号加入 Follow 集，则跳出循环。

```
    if not updated:
        break
return follow
```

4. 针对该文法，设置主函数如下：

```
first = first_set(productions)
follow = follow_set(productions, first)
ll1_productions = ll1_grammar(productions, first, follow)
print('LL(1)文法:')
for key in ll1_productions:
    print(key, '→', ' | '.join(ll1_productions[key]))
print('First集合:')
for key in first:
    print(key, ':', first[key])
print('Follow集合:')
for key in follow:
    print(key, ':', follow[key])
```

运行程序，可以得到运行结果：

```
 理\实验3\FIRST_FOLLOW_LL(1)\main.py"
LL(1)文法:
E → T E'
E' → + T E' | ε
T → int | float | double | long
B → U B'
B' → and U B' | ε
U → true | false
V → var
A → V = E | B
C → if ( B ) L else L
S → A | C
L → S ; L | S
P → L
First集合:
E : {'int', 'double', 'float', 'long'}
T : {'int', 'double', 'float', 'long'}
B : {'true', 'false'}
U : {'true', 'false'}
V : {'var'}
A : {'var', 'true', 'false'}
C : {'if'}
S : {'var', 'if'}
L : {'var', 'if'}
P : {'var', 'if'}
Follow集合:
E : {'$', '+', ';', ')', 'else'}
E' : {'$', '+', ';', ')', 'else'}
T : {'$', '+', ';', ')', 'else'}
B : {'and', '$', ';', ')', 'else'}
B' : {'and', '$', ';', ')', 'else'}
U : {'and', '$', ';', ')', 'else'}
V : {'=', '$', '+', ';', ')', 'else'}
A : {'$', '+', ';', ')', 'else'}
C : {'$', '+', ';', ')', 'else'}
S : {'$', '+', ';', ')', 'else'}
L : {'$', 'else'}
P : {'$'}

进程已结束,退出代码0
```

二、LL（1）分析表的构造和 LL（1）预测分析法的分析过程

1. 按照方法步骤中的 4，对给定 LL（1）文法的分析表进行构造

构造 LL（1）分析表，首先需要对给定的 LL（1）文法中，非终结符的 First 集和 Follow 集进行求解，这部分任务已经完成。接下来需要利用已经得到的 First 集和 Follow 集对给定 LL（1）文法的预测分析表进行构建。预测分析表是一个 M[A，a]的矩阵，其中 A 为非终结符，a 为终结符，M[A，a]存放的是一条 A 的产生式，有两种可能的值：输入字符 a 时应采用的候选式或出错标志。具体的构建过程可以按照以下步骤进行：

①对文法 G 的每个产生式 A→α执行第②、③步；

②对每个终结符 a∈First(α)，把 A→α加到 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剩余所有未定义的 M[A，a]表示出错标志 ERROR。

这一部分代码及解析如下：

```
# 初始化
inputSet = set()  # 文法符号集合
name_list = infinite.keys()  # 非终结符列表
table = {}  # 空的分析表
# 遍历所有非终结符和它们的First集和Follow集，把它们加入inputSet中
for name in name_list:
    for s in infinite[name].FIRST.keys():
        inputSet.add(s)
    for s in infinite[name].FOLLOW:
        inputSet.add(s)
inputSet.discard('ε')  # 删除inputSet中的空串

# 遍历所有文法符号，并将它们加入到分析表中
for s in inputSet:
    for name in name_list:
        table.setdefault(name, dict())
        table[name].setdefault(s, '')
        if s in infinite[name].FIRST.keys():  # 如果在内则直接将对应值放入表
            table[name][s] = infinite[name].FIRST[s]
        elif 'ε' in infinite[name].FIRST.keys() and s in infinite[name].FOLLOW:
            table[name][s] = 'ε'
        else:
            table[name][s] = 'ERROR'

# 输出第一行，即所有终结符号
for s in inputSet:
    print('{:<15}'.format(s), end="|")
print()
# 输出分析表的剩余部分
for name in table.keys():
    for s in inputSet:
        print('{:<15}'.format(str(table[name][s])),end="|")
    print()
```

2. 给定字符串，给出字符串的预测分析过程

首先对构建预测分析表的方法进行修正：在进行语法分析时，除了考虑不发生异常的情况外，还需要对同步符号情况进行记录，因此可以在遍历文法符号的过程中，添加一步判断对同步符号进行记录。

```
elif 'ε' not in infinite[name].FIRST.keys() and s in infinite[name].FOLLOW:
    table[name][s] = 'synch'  # 同步符号
```

将得到的预测分析表和待分析的字符串作为参数传入语法分析函数。在该函数中定义分析过程中需要用到的数据结构并初始化：

```
stack = []  # 建栈
name_list = list(table.keys())
actions = []  # 记录分析过程中的动作
temp_stack = []  # 记录每一个分析步骤栈的状态
temp_express = []  # 记录每一个分析步骤输入串的状态
create_uses = []  # 记录每一个步骤使用的产生式
stack.append('#')  # '#'号压入栈
stack.append(name_list[start])
actions.append('初始化')
temp_stack.append(list(stack))
temp_express.append(express)
create_uses.append(' ')
```

设置循环，在循环体中，依次读取输入表达式中的字符。如果栈顶元素为终结符，则将其与输入字符逐一匹配，如果匹配成功，弹出栈顶元素并读取下一输入字符。

```
if X.islower() or X in op or X == '#':  # 处理栈顶为终结符的情况
    if X == '#':
        actions.append("LL(1)分析结束")
        temp_express.append(express[i:])
        temp_stack.append(list(stack))
        create_uses.append("#")
        break
    else:  # 两个终结符匹配
        actions.append('GET NEXT')
        temp_stack.append(list(stack))
        create_uses.append(' ')
        i += 1
        temp_express.append(express[i:])
```

这里需要强制输入的字符串以#结尾

如果栈顶元素为非终结符，根据预测分析表中对应的产生式进行规约操作，并将产生式中的符号压入栈中。

```
elif X.isupper():  # 处理栈顶为非终结符的情况
    if a not in op and not a.isalnum() and a != '#':
        actions.append("ERROR" + "跳过" + "{}".format(a))  # 进入下一个
        stack.append(X)  # 将X中的元素重新放回
        temp_express.append(express[i:])
        temp_stack.append(list(stack))
        create_uses.append(" ")
        i += 1  # 跳过当前元素

else:
    if a.isalnum():  # 转换
        a = 'i'
    if table[X][a] == 'ERROR':  # 此时为ERROR则需要进行错误处理
        actions.append("ERROR" + "跳过" + "{}".format(a))  # 进入下一个
        stack.append(X)  # 将X中的元素重新放回
        temp_express.append(express[i:])
        temp_stack.append(list(stack))
        create_uses.append(" ")
        i += 1  # 跳过当前元素
    elif table[X][a] == 'synch':
        actions.append("synch" + "弹出" + "{}".format(X))
        temp_stack.append(list(stack))
        create_uses.append(" ")
        temp_express.append(express[i:])

    else:
        create = ''.join(table[X][a])
        if table[X][a] != 'ε':
            actions.append("POP,PUSH({})".format(''.join(reversed(create))))
            create_uses.append(X + "->" + ''.join(table[X][a]))
            for s in reversed(create):
                stack.append(s)
            temp_stack.append(list(stack))
            temp_express.append(express[i:])
        else:
            actions.append("POP")
            create_uses.append(X + "->" + ''.join(table[X][a]))
            temp_stack.append(list(stack))
            temp_express.append(express[i:])
```

过程中对使用过的产生式以及栈的状态进行记录，以列表形式进行存储，方便输出观察。

3. 给定文法如下：

```
E->TG
G->+T|-TG|ε
T->FS
S->*FS|/FS|ε
F->(E)|i
```

运行程序，可以得到文法对应的预测分析表为：

```
(           |/          |-          |)          |i          |*          |#          |+          |
['TG']      |ERROR      |ERROR      |ERROR      |['TG']     |ERROR      |ERROR      |ERROR      |
ERROR       |ERROR      |['-TG']    |ε          |ERROR      |ERROR      |ε          |['+T']     |
['FS']      |ERROR      |ERROR      |ERROR      |['FS']     |ERROR      |ERROR      |ERROR      |
ERROR       |['/FS']    |ε          |ε          |ERROR      |['*FS']    |ε          |ε          |
['(E)']     |ERROR      |ERROR      |ERROR      |['i']      |ERROR      |ERROR      |ERROR      |

进程已结束,退出代码0
```

给定字符串为：i+i*i#（程序中强制输入字符串以#结尾）

运行程序，可以得到其分析过程为：

```
每一个分析步骤栈的状态temp_stack: [['#', 'E'], ['#', 'G', 'T'], ['#', 'G', 'S', 'F'], ['#', 'G', 'S', 'i'], ['#', 'G', 'S'], ['#', 'G'], ['#', 'T',
  '+'], ['#', 'T'], ['#', 'S', 'F'], ['#', 'S', 'i'], ['#', 'S'], ['#', 'S', 'F', '*'], ['#', 'S', 'F'], ['#', 'S', 'i'], ['#', 'S'], ['#'], []]
每一个分析步骤输入串的状态temp_express: ['i+i*i#', 'i+i*i#', 'i+i*i#', 'i+i*i#', '+i*i#', '+i*i#', '+i*i#', 'i*i#', 'i*i#', 'i*i#', '*i#', '*i#',
 'i#', 'i#', '#', '#', '#']
每一个步骤使用的产生式create_uses: [' ', 'E->TG', 'T->FS', 'F->i', ' ', 'S->ε', 'G->+T', ' ', 'T->FS', 'F->i', ' ', 'S->*FS', ' ', 'F->i', ' ',
 'S->ε', '#']
栈的状态actions: ['初始化', 'POP,PUSH(GT)', 'POP,PUSH(SF)', 'POP,PUSH(i)', 'GET NEXT', 'POP', 'POP,PUSH(T+)', 'GET NEXT', 'POP,PUSH(SF)',
 'POP,PUSH(i)', 'GET NEXT', 'POP,PUSH(SF*)', 'GET NEXT', 'POP,PUSH(i)', 'GET NEXT', 'POP', 'LL(1)分析结束']
```

三、语法树的构造和预测

参照方法步骤中的 6，对预测分析过程的语法树进行构造。

1. 在已有预测分析表并能够得到给定字符串的预测分析过程的前提下，构建语法树的整体思路为：利用一个栈和一个列表来维护语法树，栈用于存储每个节点的指针，列表用于存储每个节点的属性信息，两者通过下标对应。在遇到终结符和非终结符时，创建新节点，将其指针入栈并将节点信息添加到列表中；当遇到产生式时，从栈中弹出产生式右部的符号并将其添加为新节点的子节点，同时更新父节点信息；当遇到空串时，不进行任何操作。最后分析成功时返回语法树即可。

2. 具体实现过程如下：

2.1. 初始化一个栈，并将开始符号压入栈，并在输入的字符串末尾添加结束符。

```
stack = ['$S']
input_str += '$'
```

2.2. 设置循环进行语法分析，当栈不为空且输入字符串未遍历结束时，不断取出栈顶元素和输入字符串当前位置的字符：

```
while len(stack) > 0 and i < len(input_str):
    top = stack[-1]
    a = input_str[i]
```

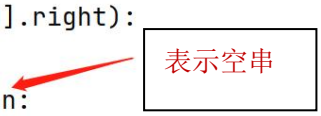
然后进行以下判断：

如果栈顶元素和当前字符相同，则弹出栈顶元素，同时移动到下一个字符。

```
if top == a:
    stack.pop()
    i += 1
```

如果栈顶元素是非终结符且预测分析表中存在对应的产生式，则使用产生式将栈顶元素替换未右部符号串。同时根据产生式构造语法树节点，并将其压入栈中。如果预测分析表中不存在对应的产生式，则抛出语法错误异常。

```
elif top in predict_table and a in predict_table[top]:
    productions = predict_table[top][a]
    if len(productions) > 0:
        stack.pop()
        for symbol in reversed(productions[0].right):
            stack.append(symbol)
        if productions[0].right[0] != epsilon:
            node = Node(productions[0].left, [])
            for _ in range(len(productions[0].right)):
                node.children.append(Node(stack.pop(), []))
            stack.append(node)
        else:
            stack.append(Node(productions[0].left, []))
    else:
        raise SyntaxError(f"Unexpected token: {a}")
else:
    raise SyntaxError(f"Unexpected token: {a}")
```



如果栈顶元素和当前字符都不是终结符，抛出语法错误。最后返回栈中的最后一个元素，未语法树的根节点。

```
    else:
        raise SyntaxError(f"Unexpected token: {a}")
return stack[0]
```

3. 当给定文法为：

```
E->T|E+T
T->F|T*F
F->(E)|i
```

构建字符串(i+i)*i 的语法树，可以得到运行结果为：

```
(S (E (T (F ( ( E ) ) ) * (F (i)))))
```

进程已结束,退出代码0

实验结论：

1. 为了验证你第一部分编写的程序是准确的，你设计了什么测试数据进行测试，得到的结果如何。

1.1. 程序的修正：

在上述第一部分的实现过程中，针对题目所给的文法的特殊性进行了设计和实现。要将求解方法应用到更多文法中，需要修改文法的处理方法：

①把文法存储在文件中（要求文法符合常规定义，不出现形如 C→if (B) L else L 的产生式），然后读取文件每一行，利用正则表达式对每一个产生式进行匹配：

```
express = f.read()
regex = "[A-Z]->[a-zA-Z|*+-/(){}ε]+"
res = re.findall(regex, express)
if len(res) == len(express.split('\n')):
    return res  # 说明输入的文法合法，否则说明输入的文法中有不匹配项
```

②然后对文法进行消除左递归并提取公共左因子。

消除左递归核心代码及解释如下：

```
# 初始化，将每个非终结符对应的产生式都存储在一个 Infinite 类型的对象中，
# 并为每个非终结符生成一个唯一的新名称
infinite, name_get = init(expresses)
# 遍历每个非终结符，尝试消除直接左递归，将其转化为等价的非左递归产生式
name_list = list(infinite.keys())
for i in range(0, len(name_list)):
    infinite_i = infinite[name_list[i]]
    for j in range(0, i):
        infinite_j = infinite[name_list[j]]
        for equal in infinite_i.equalList:
            if equal[0] == infinite_j.name:
                if len(equal)>1:
                    for j_equal in infinite_j.equalList:
                        new_equal = j_equal + equal[1:]
                        infinite_i.equalList.append(new_equal)
                    infinite_i.equalList.remove(equal)
```

```
# 检查是否存在直接左递归
new_name = name_get[0]  # 添加新非终结符
target = []
flag = False
for k in range(len(infinite_i.equalList)):
    if infinite_i.equalList[k][0] == infinite_i.name:  # 消除直接左递归
        flag = True
        for j in range(0, len(infinite_i.equalList)):
            if infinite_i.equalList[j][0] != infinite_i.name:
                # 对于不是左递归的后加符号
                infinite_i.equalList[j] += new_name
        target.append(infinite_i.equalList[k])
        infinite_i.equalList.remove(infinite_i.equalList[k])  # 删除左递归

# 为 A 创建新的非终结符 A'，并将所有以 A 开头的产生式 A -> A alpha_1 | A alpha_2 | ... 转化为
# A -> gamma_1 A' | gamma_2 A' | ...，其中 gamma_i 是产生式 A -> alpha_i beta 的后缀，
# A' 添加一个空产生式 A' -> ε，并将 A' 添加到文法中
if flag:
    infinite_new = Infinite(new_name)
    for equ in target:
        equ_temp = equ[1:] + new_name
        infinite_new.equalList.append(equ_temp)
    infinite_new.equalList.append('ε')
    infinite[new_name] = infinite_new
    name_get.remove(new_name)
```

提取公共左因子核心代码及解释如下：

```
# 遍历每个非终结符，计算其 FIRST 集合中的终结符
name_list = list(infinite.keys())
for name in name_list:
    first_set = set()
    for equal in infinite[name].equalList:
        first_set.add(equal[0])

# 遍历 FIRST 集合中的终结符，检查是否存在公共左因子
for first in first_set:
    same_equal = [equal for equal in infinite[name].equalList if equal[0] == first]
```

```
if len(same_equal) >1:
    # 说明存在公共左因子，则采取提取公共左因子操作
    new_name = name_get[0]
    new_infinite = Infinite(new_name)
    name_get.remove(new_name)
    same = same_equal[0][0]
    # 遍历所有具有公共左因子的产生式，将公共左因子提取出来，并将剩余部分存储在新的非终结符中
    for equ in same_equal:
        new_infinite.equalList.append(equ[1:])
        infinite[name].equalList.remove(equ)
    # 将新的非终结符添加到文法中，并将原产生式改写为不包含公共左因子的形式
    infinite[name].equalList.append(same+new_name)
    infinite[new_name] = new_infinite
```

③对文法 First 集和 Follow 集的求解思路不变，这里不重复说明。

1.2. 设置正例进行测试：

```
E->TG
G->+T|-TG|ε
T->FS
S->*FS|/FS|ε
F->(E)|i
```

运行结果：

```
该文法是LL（1）文法
```

1.3. 设置违例进行测试：

```
S->AB
S->bC
A->ε
A->b
B->ε
B->aD
C->AD
D->aS
D->c
```

运行结果：

```
各个非终结符的FIRST集：
S:{'ε': ['AB'], 'b': ['AB', 'AB', 'bC']}
A:{'ε': ['ε'], 'b': ['b']}
B:{'ε': ['ε'], 'a': ['aD']}
C:{'ε': ['εD'], 'b': ['bD']}
D:{'a': ['aS'], 'c': ['c']}
    FOLLOW_next(infinite, name, get_record, name_list)
```

发现程序运行至 Follow 集的过程中出现报错，这是因为我的程序中 Follow 集的求解在具体实现时以 LL（1）文法为前提。虽也可以对违例进行验证，但此处仍有待完善。

2. 为了验证你第二部分编写的程序是准确的，你设计了什么测试数据进行测试，得到的结果如何。

2.1. 设置待分析的字符串为：i+i*i#

运行结果：

```
每一个步骤使用的产生式create_uses: [' ', 'E->TG', 'T->FS', 'F->i', ' ', 'S->ε', 'G->+T', ' ', 'T->FS', 'F->i', ' ', 'S->*FS', ' ', 'F->i', ' ',
 'S->ε', '#']
栈的状态actions: ['初始化', 'POP,PUSH(GT)', 'POP,PUSH(SF)', 'POP,PUSH(i)', 'GET NEXT', 'POP', 'POP,PUSH(T+)', 'GET NEXT', 'POP,PUSH(SF)',
 'POP,PUSH(i)', 'GET NEXT', 'POP,PUSH(SF*)', 'GET NEXT', 'POP,PUSH(i)', 'GET NEXT', 'POP', 'LL(1)分析结束']
```

2.2. 设置待分析的字符串为：i+i*)#，对同步符号异常情况进行处理。

运行结果：

```
每一个分析步骤输入串的状态temp_express: ['i+i*)#', 'i+i*)#', 'i+i*)#', 'i+i*)#', '+i*)#', '+i*)#', '+i*)#', 'i*)#', 'i*)#', 'i*)#', '*)#', '*)#',
 ')#', ')#', ')#', ')#']
每一个步骤使用的产生式create_uses: [' ', 'E->TG', 'T->FS', 'F->i', ' ', 'S->ε', 'G->+T', ' ', 'T->FS', 'F->i', ' ', 'S->*FS', ' ', ' ', 'S->ε', '#']
栈的状态actions: ['初始化', 'POP,PUSH(GT)', 'POP,PUSH(SF)', 'POP,PUSH(i)', 'GET NEXT', 'POP', 'POP,PUSH(T+)', 'GET NEXT', 'POP,PUSH(SF)',
 'POP,PUSH(i)', 'GET NEXT', 'POP,PUSH(SF*)', 'GET NEXT', 'synch弹出F', 'POP', 'LL(1)分析结束']
```

2.3. 设置待分析的字符串为：i+i&#，对错误情况进行处理。

运行结果：

```
每一个分析步骤栈的状态temp_stack: [['#', 'E'], ['#', 'G', 'T'], ['#', 'G', 'S', 'F'], ['#', 'G', 'S', 'i'], ['#', 'G', 'S'], ['#', 'G'], ['#', 'T',
  '+'], ['#', 'T'], ['#', 'S', 'F'], ['#', 'S', 'i'], ['#', 'S'], ['#', 'S'], ['#'], []]
每一个分析步骤输入串的状态temp_express: ['i+i&#', 'i+i&#', 'i+i&#', 'i+i&#', '+i&#', '+i&#', '+i&#', 'i&#', 'i&#', 'i&#', '&#', '&#', '#', '#']
每一个步骤使用的产生式create_uses: [' ', 'E->TG', 'T->FS', 'F->i', ' ', 'S->ε', 'G->+T', ' ', 'T->FS', 'F->i', ' ', ' ', 'S->ε', '#']
栈的状态actions: ['初始化', 'POP,PUSH(GT)', 'POP,PUSH(SF)', 'POP,PUSH(i)', 'GET NEXT', 'POP', 'POP,PUSH(T+)', 'GET NEXT', 'POP,PUSH(SF)',
 'POP,PUSH(i)', 'GET NEXT', 'ERROR跳过&', 'POP', 'LL(1)分析结束']
```

3. 为了验证你第三部分编写的选做程序是准确、有用的，你设计了什么测试数据进行测试，得到的结果如何。

3.1. 正例已经在实验过程中给出。

3.2. 接下来对语法分析错误处理能力进行测试：

给定一个文法如下（NUM 表示数字）：

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

给出违例字符串为：2+3*(4-1。在这个字符串中，缺少右括号，因此会抛出 SyntaxError 异常，指出输入字符串存在语法错误，无法进行语法分析。运行结果：

```
Traceback (most recent call last):
  File "[redacted]编译原理\实验3\LL(1)_analysis\tree.py", line 94, in <module>
    root = parse(predict_table, start_symbol, '2+3*(4-1')
  File "[redacted]编译原理\实验3\LL(1)_analysis\tree.py", line 47, in parse
    raise SyntaxError(f"Unexpected token: {a}")
SyntaxError: Unexpected token: 2
```

心得体会：

通过本次实验，对自顶向下的语法分析程序设计进行了学习和实现。学会如何根据所需和所求，设计程序对文法进行处理、求解文法非终结符的 First 集和 Follow 集；并根据要求的增加，不断在原程序的基础上进行完善，利用已知求预测分析表，构建语法分析过程并采用合适的数据结构进行表示。

对自顶向下的语法分析程序的理解：它是一种基于上下文无关文法的语法分析方法，是自上而下递归地对输入符号串进行分析的过程。程序从文法的起始符号开始，根据文法的规则和输入符号串的符号，逐步推导出所有可能的句子。
指导教师批阅意见：   成绩评定： 指导教师签字：蔡树彬 2023 年 5 月 25 日
备注：