自顶向下的语法分析

实验内容

给定下面的文法

```
E  → T E'
E' → + T E'| e
T  → F T'
T' → * F T'| e
F  → ( E ) | id
```

请使用递归下降法生成语法树，并将语法树打印成字符串形式，语法树的打印成字符串使用中括号嵌套的形式.

例如id + id的语法树可以为字符串:

`[E [T [F [id]] [T' [e]]] [E' [+] [T [F [id]][T'[*][F [id]] [T'[e]]]] [E'[e]]]]`

语言与调用方法

python3.9

```
python demo.py
```

实验步骤

实验原理

在定义语法分析程序的时候，每一个非终极符都**定义成一个过程或者函数**:

1. 每棵子树都是以根节点的非终极符推导出来的短语
2. 可以考虑每个非终极符构造一个函数，去匹配子树的叶节点

同时为了保证推导的唯一性，**对文法的要求**与LL(1)文法相同:

1. 文法不能是直接左递归的
2. 以一个非终结符为左部的任意规则的predict集，交集为空

1. 非终结符的定义

由于每个非终结符first集合中的元素都**唯一对应着一个文法表达式**，所以我们可以根据接下来的字符，判断非终结符应该由哪个表达式进行转换

例如:

```
def E_(e_):
    global i
    if sentence_list[i] == '+':
        i += 1
        print("E' → +TE'")
        plus, t, e_1 = Node("+"), Node("T"), Node("E'")
        e_.child += [plus, t, e_1]
        T(t)
        E_(e_1)
    else:
        print("E'->e")
        e_.child.append(Node('e'))
```

其中，全局变量`i`指向原句`sentence_list`中待分析的下一个词

当我们遇到`E'`时，我们有两种转换方式：

1. **E' -> e**
2. **E' -> + T E'**

所以我们判断`i`所指向的是否为`+`号：

如果是，那么创建三个新结点，分别对应`+ T E`，将它们依次递归调用，并放入当前结点的孩子中

如果不是，那么只需要将一个新的空结点放入当前结点的孩子即可，不需要继续递归下去

每一个非终结符都有一个与之类似的函数：

```
def E(e):
    print("E → TE'")
    t, e_ = Node('T'), Node("E'")
    e.child += [t, e_]
    T(t)
    E_(e_)

def E_(e_):
    global i
    if sentence_list[i] == '+':
        i += 1
        print("E' → +TE'")
        plus, t, e_1 = Node("+"), Node("T"), Node("E'")
        e_.child += [plus, t, e_1]
        T(t)
        E_(e_1)
    else:
        print("E'->e")
        e_.child.append(Node('e'))

def T(t):
    print("T → F T'")
    f, t_ = Node("F"), Node("T'")
    t.child += [f, t_]
    F(f)
    T_(t_)

def T_(t_):
    global i
    if sentence_list[i] == "*":
```

```
        i += 1
        print("T\' → * F T\' ")
        mul, f, t_1 = Node("*"), Node("F"), Node("T'")
        t_.child += [mul, f, t_1]
        F(f)
        T_(t_1)
    else:
        print("T' → e")
        t_.child+=[Node("e")]

def F(f):
    global i
    if sentence_list[i] == "(":
        i += 1
        print("F → (E)")
        lb, e, rb = Node("("), Node("E"), Node(")")
        f.child += [lb, e, rb]
        E(e)
        i += 1
    elif sentence_list[i] == "id":
        i += 1
        print("F → id")
        id = Node("id")
        f.child += [id]
    else:
        print("Error!Check the sentence!")
        exit()
```

2. 树上结点的定义

将树上结点定义成新的类**Node**，其中保存对应的非终结符，并写成构造函数方便调用

同时，定义一个**to_tree()**函数，用于获取当前结点的形式化格式，便于在网站检验

```
class Node:
    def __init__(self, ch=None):
        self.ch = ch
        self.child = []

    def to_tree(self):
        if len(self.child)!=0:
            res=self.ch
            for i in self.child:
                res+=i.to_tree()
            return "["+res+"]"
        else:
            return "["+self.ch+"]"
```

3. error的处理

主要有两种错误：

1. **匹配不上相应的非终结符**
2. **匹配结束后，仍然剩余字符**

对于本文法，发现1错误只可能出现在**F → (E)** 规则中，于是只需要在**F(f)**函数中添加如下判断即可：

```
def F(f):
    if sentence_list[i] == "(": ...
    elif sentence_list[i] == "id": ...
    else:
        print("Error!Check the sentence!")
        exit()
```

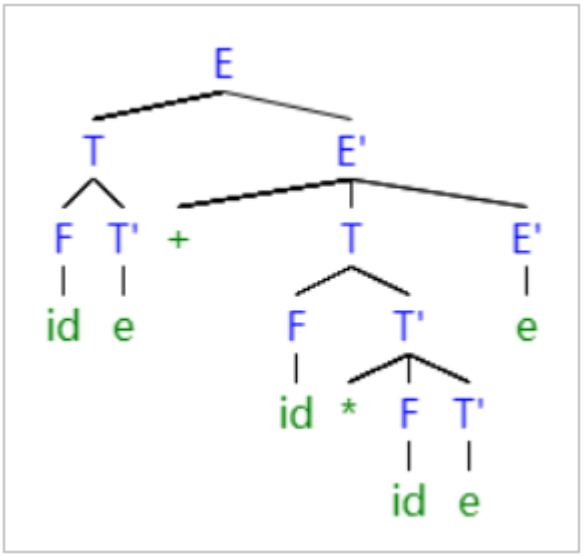
对于2错误，我们可以在sentence末尾加上#符来判断是否匹配完整个句子：

```
sentence_list = sentence.split() + ["#"]
e = Node('E')
E(e)
if sentence_list[i]!="#":
    print("Error!Check the sentence!")
    exit()
```

实验结果

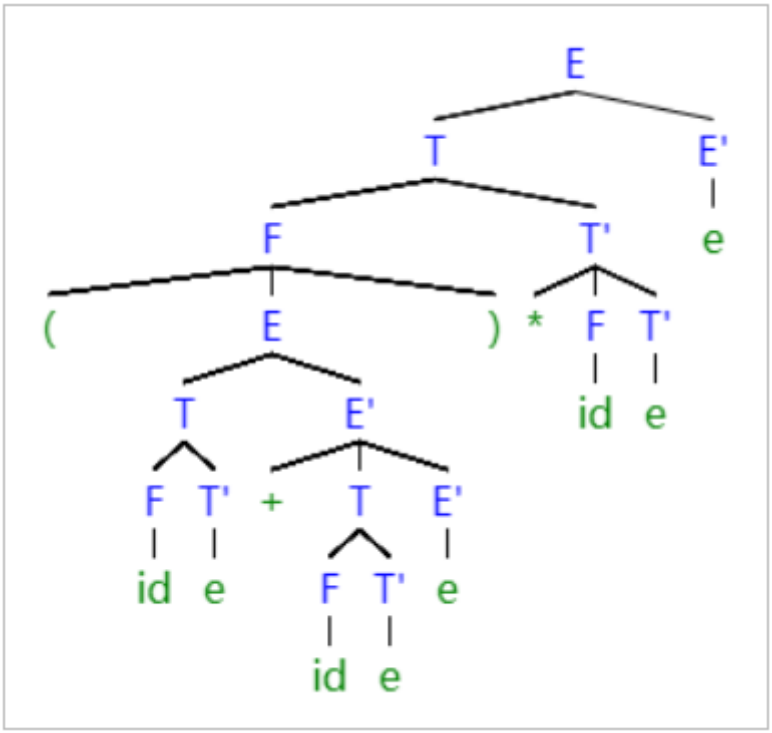
1. id + id * id

```
E → TE'
T → F T'
F → id
T' → e
E' → +TE'
T → F T'
F → id
T' → * F T'
F → id
T' → e
E'->e
[E[T[F[id]][T'[e]]][E'[+][T[F[id]][T'[*][F[id]][T'[e]]]][E'[e]]]]
```



2. (id + id) * id

```
E → TE'
T → F T'
F → (E)
E → TE'
T → F T'
F → id
T' → e
E' → +TE'
T → F T'
F → id
T' → e
E'->e
T' → * F T'
F → id
T' → e
E'->e
[E[T[F[(][E[T[F[id]][T'[e]]][E'[+][T[F[id]][T'[e]]][E'[e]]]][)]][T'[*][F[id]][T'[e]]]][E'[e]]]
```



3. id * id (id)

```
E → TE'
T → F T'
F → id
T' → * F T'
F → id
T' → e
E'->e
Error!Check the sentence!
```

4. id * * id

```
E → TE'
T → F T'
F → id
T' → * F T'
Error!Check the sentence!
```