院系	年级专业	姓名	学号	实验日期
计算机学院	2019计科	吴家隆	1915404063	2021.9.27

编程语言：***python3.9***

自顶向下的语法分析

实验内容

给定下面的文法:

$$E \to TG$$
$$G \to +TG|e$$
$$T \to FS$$
$$S \to *FS|e$$
$$F \to (E)|id$$

- 输入字符串后使用**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。
- 在 http://mshang.ca/syntree/ 将字符串形式生成树状图, 以检验是否正确。

实验步骤

功能

利用函数之间的递归调用模拟语法树自上而下的构造过程。

前提条件

改造文法：消除二义性、消除左递归、提取左因子，判断是否为LL（1）文法。

题中文法符合上述条件

设计思想及算法

为G的每个非终结符号U构造一个递归过程,不妨命名为U。

U的产生式的右边指出这个过程的代码结构：

1. 若是终结符号，则和向前看符号对照，若匹配则向前进一个符号；否则出错。
2. 若是非终结符号，则调用与此非终结符对应的过程。当A的右部有多个产生式时,可用选择结构实现。

- 对于每个非终结符号U→ X1 ｜ X2 |...| Xn处理的方法如下

```
U()
{
    Read(ch);//当前符号
    If ch in First(X1)    调用X1的子程序;
    else if ch in First(X2)    调用X2的子程序;
    else error()
}
```

- 对于每个右部X1→Y1 Y2 ... Yn的处理架构如下：

  调用Y1的子程序;

调用Y2的子程序；

…

调用Yn的子程序

- 如果右部为空，则不处理。
- 对于右部中的每个符号Yi

  ① 如果Yi为终结符号：

```
if(Yi == 当前的符号)
{
    Read(ch);
    return;
}
else error()
```

  ② 如果Yi为非终结符号，直接调用相应的子过程Yi()

代码解释

在上述抽象的算法中，有ch表示当前的符号，我们在python程序中定义全局变量为当前的位置。

```
global i
i = 0
```

同样不使用error()函数，而是用全局变量flag表示执行过程中是否出错，并在最后输出时判断flag是否仍未True。

```
global flag
flag = True
```

由于最后要输出的是语法树的字符形式，我们定义了以下类，表示Node

```
class Node:
    def __init__(self, val=None):
        self.val = val
        self.l_child = []

    def add_child(self, node):
        self.l_child.append(node)
```

因为在生成语法中，可能有大于2的节点，所以Node为n叉树的节点。

接下来的有如下的函数表示递归过程

```
def E(p):
    print("E->TG")
    T1 = Node('T')
    G1 = Node('G')
    p.add_child(T1)
    p.add_child(G1)
    T(T1)
    G(G1)


def T(T):
    print("T->FS")
    F1 = Node('F')
    S1 = Node('S')
    T.add_child(F1)
```

```
    T.add_child(S1)
    F(F1)
    S(S1)


def G(g):
    global i
    if src[i] == "+":
        i += 1
        print("G->+TG")
        plus = Node("+")
        T1 = Node("T")
        G1 = Node("G")
        g.add_child(plus)
        g.add_child(T1)
        g.add_child(G1)
        T(T1)
        G(G1)
    else:
        print("G->e")
        e = Node("e")
        g.add_child(e)


def S(p):
    global i
    if src[i] == "*":
        print("S->*FS")
        time = Node("*")
        F1 = Node("F")
        S1 = Node("S")
        p.add_child(time)
        p.add_child(F1)
        p.add_child(S1)
        i += 1
        F(F1)
        S(S1)
    else:
        print("S->e")
        e = Node("e")
        p.add_child(e)


def F(p):
    global i
    global flag
    if src[i] == "(":
        i += 1
        left = Node("(")
        E1 = Node("E")
        E(E1)
        p.add_child(left)
        p.add_child(E1)
        if src[i] == ")":
            i += 1
            print("F->(E)")
            right = Node(")")
            p.add_child(right)
        else:
            flag = False
    elif src[i] == "id":
        print("F->id")
```

```
            id = Node("id")
            p.add_child(id)
            i += 1
        else:
            flag = False
```

由于要生成树，所以在递归过程中，要传入当前位置的结点，根据转化对该节点进行插入节点的操作。

要注意处理好终结符号和非终结符号的区别。

```
src = "( id + id ) *  id"
src = src.split() + ["#"]
root = Node('E')
E(root)
global res
res = ''


def put2str(node):
    global res
    if node:
        res += node.val
    if node.l_child:
        for i in node.l_child:
            res += "["
            put2str(i)
            res += "]"


if src[i] == "#" and flag:
    put2str(root)
    print("[" + res + "]")
else:
    print("input error!")
```

这一部分的代码是处理输入和输出。

注意输入的部分每个符号必须用空格隔开，这样方便使用split()函数进行分割，并在列表末尾加入”#“用于判断是否处理完了所有字符。

put2str(node)函数递归生成了字符串形式。

在最后输出时，要判断字符列表是否走到了最后的末尾”#“，并且在执行过程中flag没有变为False，即过程中没有出错。

实验结果

- 例子1： id+id*id

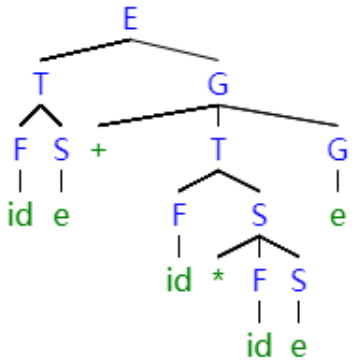
  程序输出：

```
E->TG
T->FS
F->id
S->e
G->+TG
T->FS
F->id
S->*FS
F->id
S->e
G->e
[E[T[F[id]][S[e]]][G[+][T[F[id]][S[*][F[id]][S[e]]]][G[e]]]]
```

生成的字符串：

$[E[T[F[id]][S[e]]][G[+][T[F[id]][S[*][F[id]][S[e]]]][G[e]]]]$

生成树状图验证：



- 例子2： id+id*

  程序输出：

```
E->TG
T->FS
F->id
S->e
G->+TG
T->FS
F->id
S->*FS
S->e
G->e
input error!
```

  我们看到输出了input error!的字符串，说明该字符串不合法

- 例子3 : id+(id+id)*id

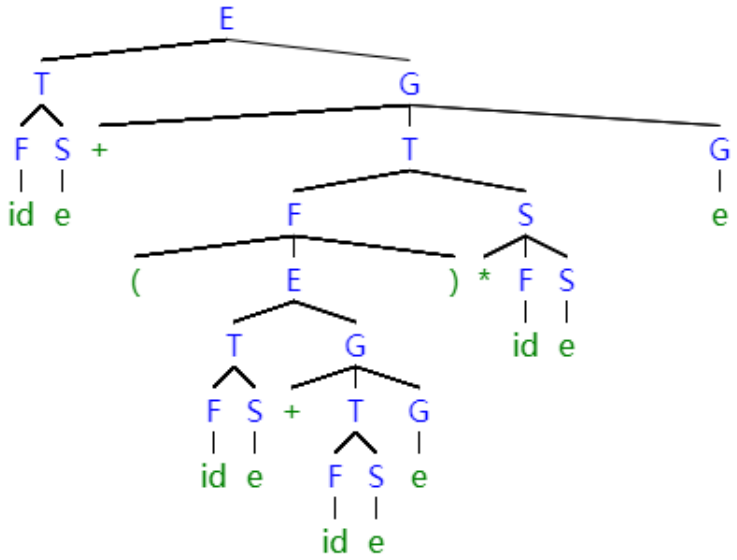
  程序输出：

```
E->TG
T->FS
F->id
S->e
G->+TG
T->FS
E->TG
T->FS
F->id
S->e
G->+TG
T->FS
F->id
S->e
G->e
F->(E)
S->*FS
F->id
S->e
G->e
[E[T[F[id]][S[e]]][G[+][T[F[(][E[T[F[id]][S[e]]][G[+][T[F[id]][S[e]]][G[e]]]][)]][S[*][F[id]][S[e]]][G[e]]]]
```

  生成的字符串：

$[E[T[F[id]][S[e]]][G[+][T[F[(][E[T[F[id]][S[e]]][G[+][T[F[id]][S[e]]][G[e]]]][)]][S[*][F[id]][S[e]]]][G[e]]]$

生成树状图验证：



以上结果均正确

- 在该程序中使用e表示空集，并假设文法中不会出现符号e和中括号.
- 为方便程序表示，将E'表示为G，将T'表示为S