编程语言的设计原理课程项目

学号：1500012204

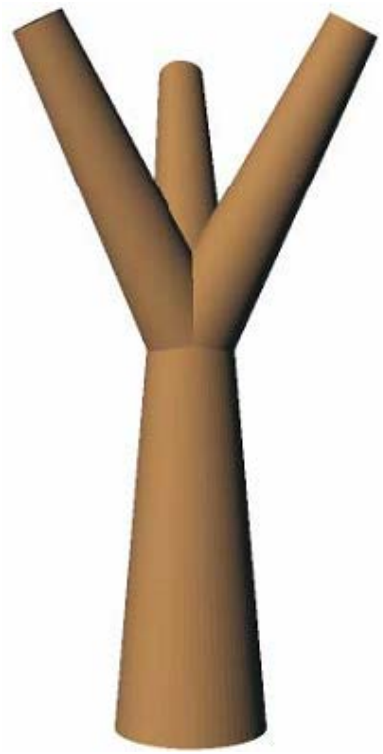
姓名：李博

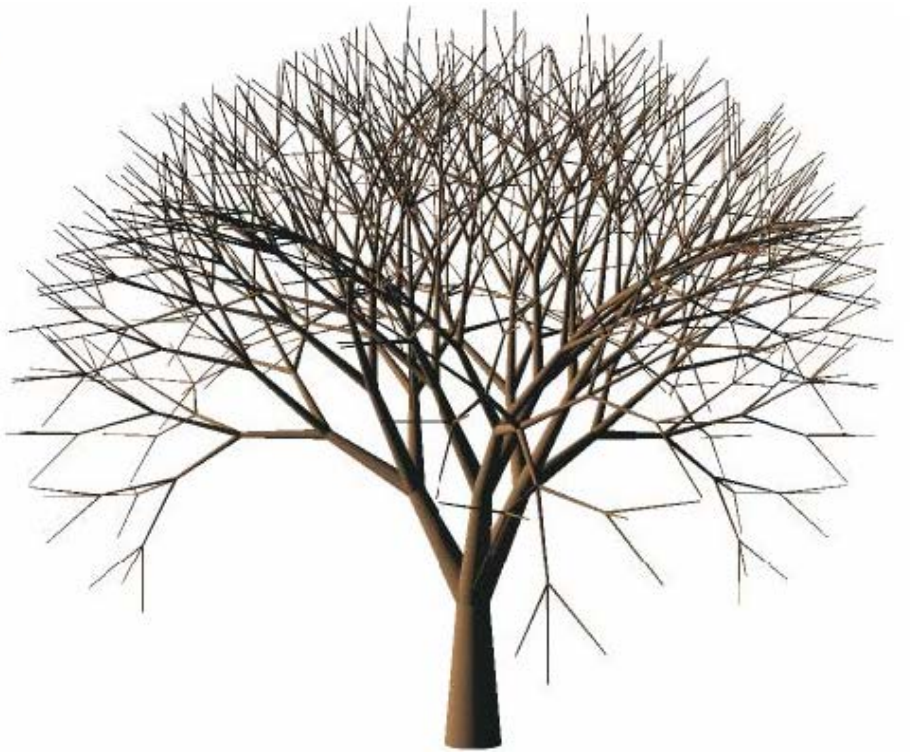
生命科学学院

计算机科学与技术双学位

背景介绍

自然界的某些物体具有自相似（self-similarity）的性质，通过研究这种性质可以方便我们对其进行表示和模拟。从结构生成的角度来说，自相似性可以看成是从父结构不断地产生子结构的过程。过程式建模（procedural modeling）就是依据这种视角，通过使用一组规则来控制生成复杂结构的方法。使用这种技术，我们可以从简单、基本的模型来生成自然界中的许多真实模型。如下图中树木中存在着自相似的三分支结构。通过不断地重复子结构，可以获得完整的树木（图片来自[1]）。

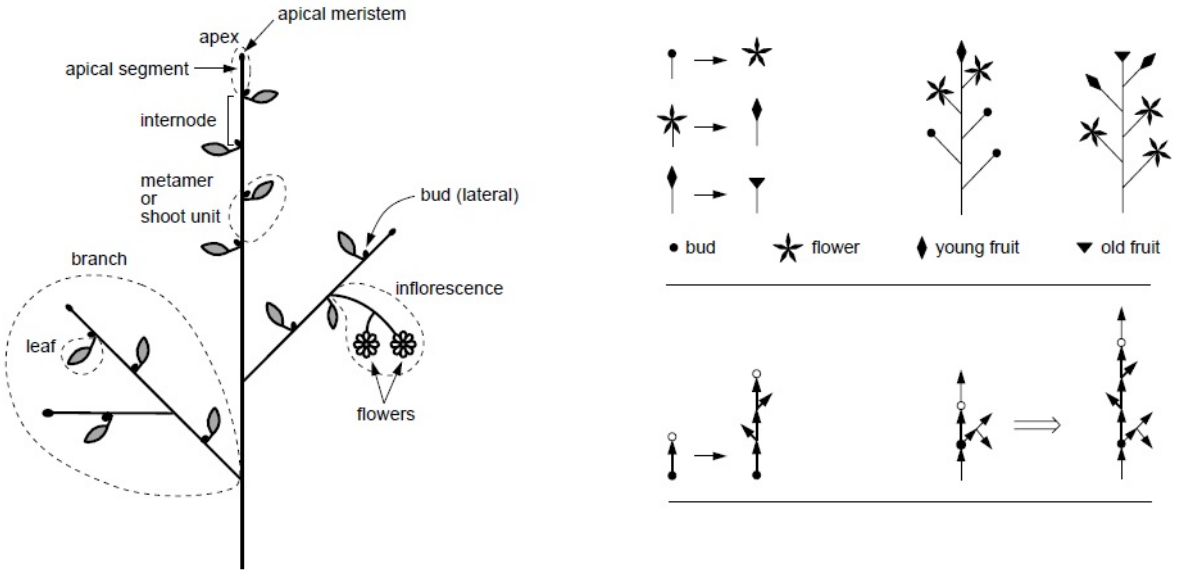




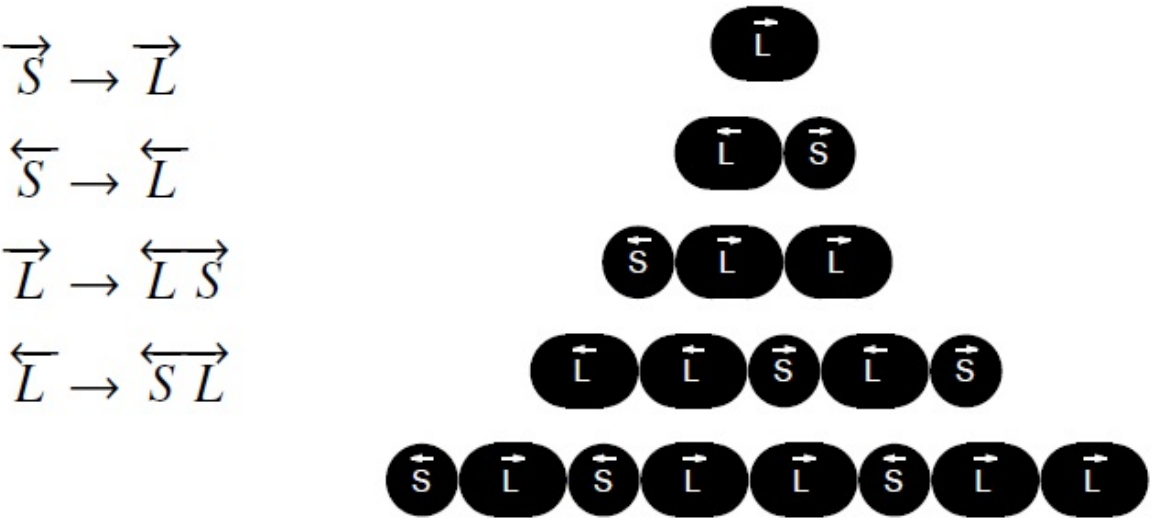
L-system就是这样一种重写系统（rewriting system）。这种系统很类似于（编程）语言的Chomsky文法系统。它使用符号来描述几何对象，并在符号上不断应用产生式（production）来衍生出新的符号，并最后把这些符号解释（interpret）成结合图形。在L-system中，这些符号的序列称为L-string。我们发现，L-string可以被视为一种描述树形结构序列化（serialization）的语言。在本项目中，我使用Python使用了一个L-system，并形式化定义了L-string上的语言。通过evaluation，可以把L-string从序列化描述转化成真正的树拓扑，并能够方便地进行图可视化。通过typing，我们可以保证这个树的描述符合一定的规范形式（regular form），以保证上下文相关（context sensitive）产生式匹配的正确性。

L-system简介

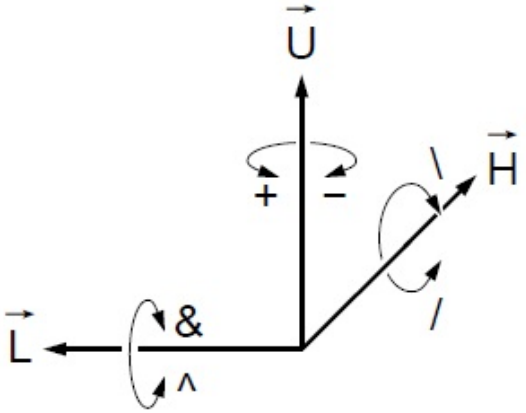
L-system是由匈牙利生物学家A. Lindenmayer最早于1968年创造的建模系统。这个系统最初是在研究一些简单多细胞丝状模式生物的发育特征中提出的[2]。后来，经过Prusinkiewicz等人的发展，L-system成为了高等植物建模的经典方法[3]。它把植物的不同部分看成模块（如芽、叶、茎、花等），用一系列符号表示这些模块。L-system通过形式语法来描述植物生长发育的形态变化（如芽分化出叶、花等）的过程，即一个模块如何被新的模块所替代的规则（见下图）。

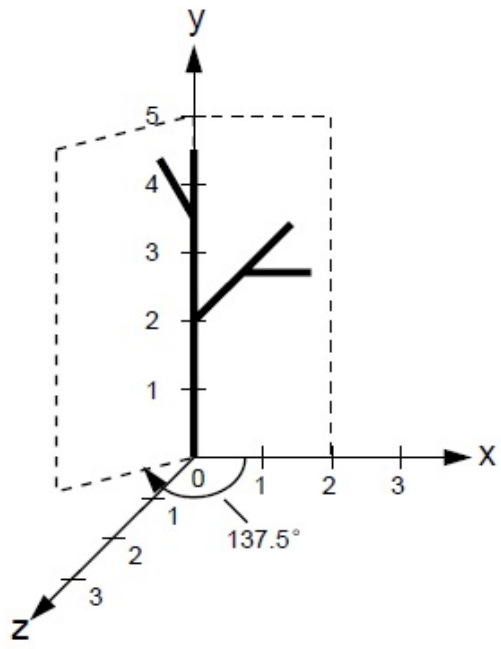


下图给出了一个简单的例子。在这个系统中，有4种符号（可能代表一些细胞）。从一个（或者一些）符号开始，不断地应用产生式，让这个符号序列（即L-string）发生变化。要注意的是，在一步中，必须同时对所有（满足条件）的符号应用产生式，这表示生物各部分都在同时发育。因此，这个重写系统有并行（parallel）的特征。而Chomsky文法中的产生式每一次只应用一条，产生了一颗递归定义的语法树。这是L-system和Chomsky文法的区别。



为了能用符号序列表示三维世界的结构，需要确定从L-string到真实模型的解释（interpret）方法。通过引入一些几何空间的变换（Transform），可以操纵在三维空间的位置。而L-string中的符号（或者说模块Module）则可以负责绘制几何体等。这种方法被称为turtle interpretation，其名称来自于LOGO玩具，含义是L-string作为指令注记就像“操纵”着一只虚拟的“海龟”，在三维空间中进行图形绘制。如下图中的F表示向前画线，通过+-等指令控制旋转，就可以画出右下角L-string所指示的图形。另外，还使用了方括号以引入分支的概念。每当在L-string中遇到‘[’符号时，说明新的分支出现了，要把当前的turtle状态（位置和角度）压入栈以保存，而之后的指令就是在新分支上执行的；每当遇到‘]’，说明该分支结束，进行出栈操作，可以回复到父分支的状态，以便继续解释父分支的其余部分，或者是从父分支衍生的其他分支。

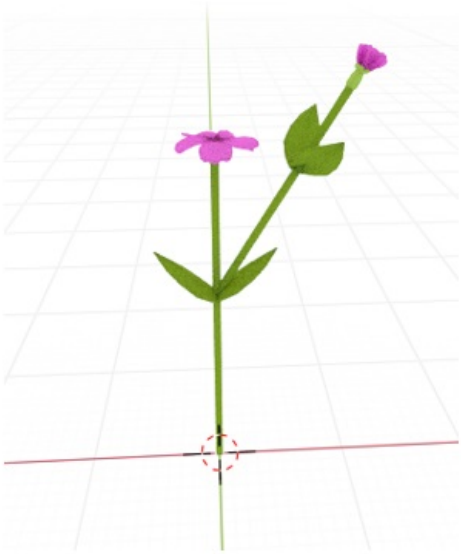


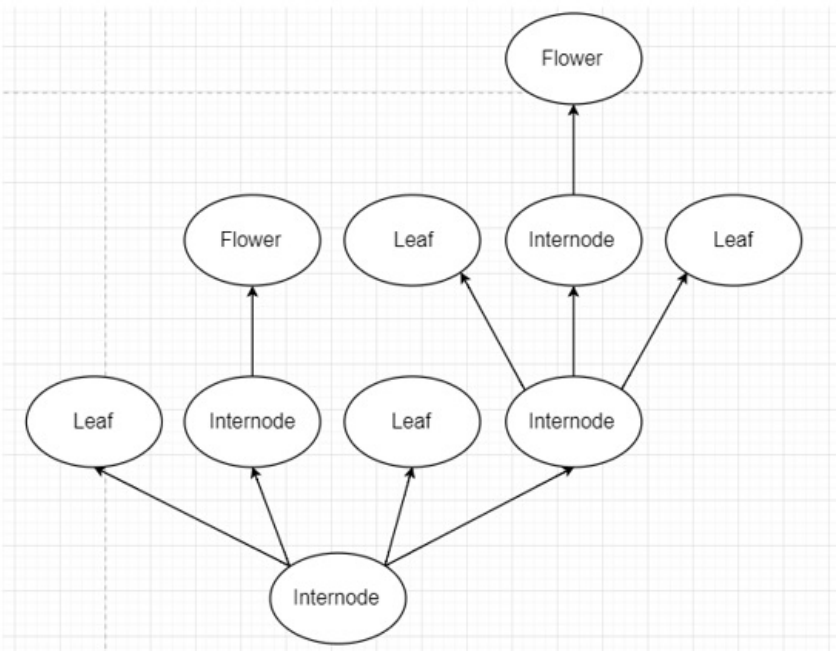


F(2)[−F[−F]F]/(137.5)F(1.5)[−F]F

L-string作为一种树的序列化语言

L-string可以视为画图命令，在从左向右读的过程中注意方括号指示的压栈、弹栈。然而，从结构而非画图指令的角度看，L-string实际上标记了一棵树的拓扑关系。前后的两个Module表示父子节点关系，而出现了方括号所包围的符号则表明出现了分支。（我用Python实现了L-system，这些方括号不再是字符，而是表示Python的list。因此，一个L-string是nested list。）比如，序列[Internode, [Leaf], [Leaf], [Internode, [Leaf], [Leaf], [Internode, Flower], Internode, Flower]表示的真实模型和拓扑关系如下图所示。

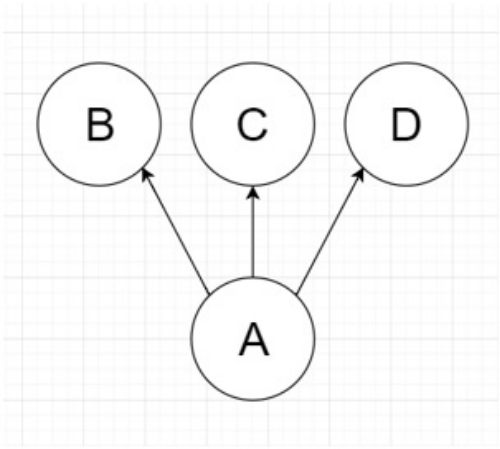




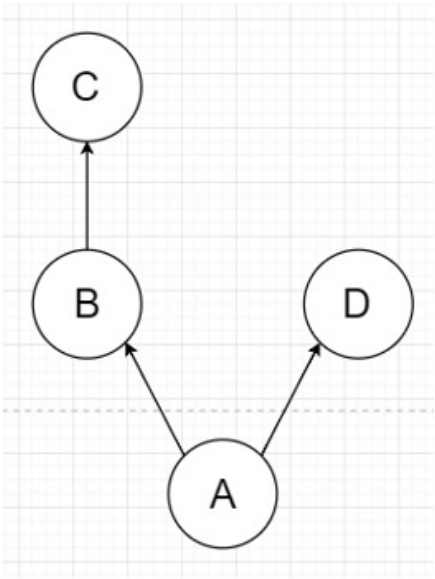
这样来看，L-string实际上是一棵树的序列化。它也可以看为一种简单的标记语言，前后关系','（有时描述省略了这个逗号）和方括号'[]'有对（子）树进行操作的语义。**对这个string进行evaluation，就是反序列化的过程，即建立真实的树拓扑的过程。**

不过，这种表示方法存在一些问题——它不是唯一的。如下图所示，两种string可能对应同一个拓扑结构。

A, [[B], C], D
A, [B], [C], D



A, [B, [C]], D
A, [B, C], D



可以看到，这种不唯一的表示是由于方括号的不规范使用造成的。我们发现，这种不规范其实可以被消除掉：

[...,[X]] -> [..., X] 在X后面该分支已经结束了，X没有必要再套括号

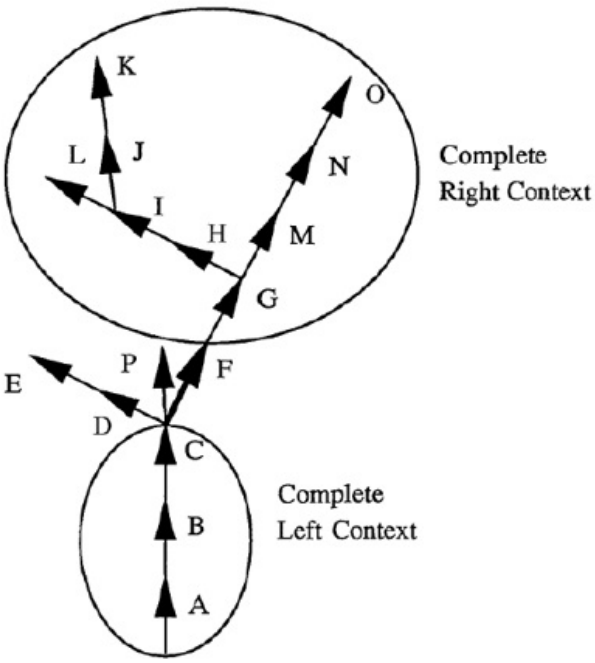
[[X], ...] -> [X], [...] 实际上是多个并列分支，他们都是X前面那个Module的子树（节点）

[[X]] -> [X] 套多层括号没有意义

这种不规范的表示一个害处在于使用上下文相关产生式的时候。这种产生式的语法是：

left_context < predecessor > right_context -> successor

以下图为例。F为predecessor，它的完整左上下文是那些所有是其祖先的节点模块，而完整右上下文是那些所有是其后代的节点模块。BC<F是能通过这个string的左上下文匹配的，F>G[H]M也能通过右上下文匹配，归根结底是pattern所体现出的树的edge在string中全部找到了。但要单纯地看string的话，不太容易判断匹配成功与否。



String:
ABC[DE][FG[HI[JK]L]MNO]P

Left: BC<F ... ✔

Right: F > G[H]M ... ✔

左上下文的匹配实现较为容易一些，因为左边的pattern序列是追溯父节点，不可能出现[]分支。如果匹配时L-string中出现了分支，直接跳过即可。但在右上下文匹配中，pattern也能出现分支。由于方括号代Python中的list，进行分支比较时需要进入到list中去比较。如果有不规范表示，如A, [[B], C], D和A, [B], [C], D，其中两个B的嵌套层数不同，这会让匹配算法的考虑变得复杂。

从对绘图状态压栈弹栈的命令角度看，不规范表示不会有什么问题。但是从树形结构上看，不规范表示会带来概念上模糊不清的问题。如在[A], [B], C中，A和B的父节点是谁？

基于以上原因，我们使用类型系统来消除不规范表示。

形式定义

Syntax

Terms

t :=

tree

module

branch

transform

[t]

t, t

说明：

（1）transform是用于空间变换的那些符号。

（2）我们的项存储在Python的list里，t,t就是相邻的两个元素，[t]就是把t放在一个list里。

（3）在evaluation前的L-string只是由module和transform组成的nested list。

（4）tree和branch是只有在evaluation过程中才会出现的项。

Values

v :=

tree

branch

transform

[v]

Types

T :=

Tree

Branch

Transform

Evaluation Rules

$$\frac{\mathrm{t}_1 \rightarrow \mathrm{t}_1'}{\mathrm{t}_1,\ \mathrm{t}_2 \rightarrow \mathrm{t}_1',\ \mathrm{t}_2} \tag{E1}$$

$$\frac{\mathrm{t}_2 \rightarrow \mathrm{t}_2'}{\mathrm{v}_1,\ \mathrm{t}_2 \rightarrow \mathrm{v}_1,\ \mathrm{t}_2'} \tag{E2}$$

$$\mathrm{module} \rightarrow \mathrm{tree} \tag{E3}$$

$$\mathrm{tree}_1,\ \mathrm{tree}_2 \rightarrow \mathrm{tree}' \tag{E4}$$

$$\mathrm{tree},\ \mathrm{transform} \rightarrow \mathrm{tree} \tag{E5}$$

$$\frac{\mathrm{t} \to \mathrm{t}'}{[\mathrm{t}] \to [\mathrm{t}']} \tag{E6}$$

$$[\mathrm{tree}_1],\ \mathrm{tree}_2 \to \mathrm{branch} \tag{E7}$$

$$[\mathrm{tree}],\ \mathrm{branch} \to \mathrm{branch}' \tag{E8}$$

$$\mathrm{tree},\ \mathrm{branch} \to \mathrm{tree}' \tag{E9}$$

说明：

（E1）（E2）保证先evaluate左边的，再做右边的。

（E3）一个module也能成为一颗树。因此，在T中不设置Module类型，module也不是value。

（E4）连接两颗树，变成一颗新树。tree是一张有向图（使用Python的networkx包来存储表示node和edge），另外还记录root以及active node。active node的作用是，记录从这颗tree发生的子树是连接到哪个node的。ree1, tree2表示把tree2的root和tree1的active node连接，从而建立两颗树的父子关系。这里为了整洁没有把这些语义写入规则列表里。

（E5）transform不应该记入module所组成的树结构。因此它碰到tree就会被忽略掉。同样类似的还有三个规则，即transform放左边，和transform被branch“吸收”的两条规则。

（E6）计算[]内部的内容。

（E7）branch的发生。branch存储了很多tree，他们都是前面某个tree的子树。[tree1], tree2从最右段产生一个branch，这个branch中暂时有这两棵树，它们是平行的分支。

（E8）branch向左遇到更多的树，把它加入平行分支中。

（E9）branch继续向左遇到tree后，会把branch中所有的分支树接到tree上。

（E7）（E8）（E9）联合保证了，类似A, [B], C这种，带方括号的项被夹在中间，才能被evaluate成tree，从而消除了不规范表示。

Typing Rules

$$\mathrm{tree} : \mathrm{Tree} \tag{T1}$$

$$\mathrm{module} : \mathrm{Tree} \tag{T2}$$

$$\mathrm{branch} : \mathrm{Branch} \tag{T3}$$

$$\mathrm{transform} : \mathrm{Transform} \tag{T4}$$

$$\frac{\mathrm{t}_1 : \mathrm{Tree},\ \mathrm{t}_2 : \mathrm{Tree}}{\mathrm{t}_1,\ \mathrm{t}_2 : \mathrm{Tree}} \tag{T5}$$

$$\frac{\mathrm{t}_1 : \mathrm{Tree},\ \mathrm{t}_2 : \mathrm{Transform}}{\mathrm{t}_1,\ \mathrm{t}_2 : \mathrm{Tree}} \tag{T6}$$

$$\frac{\mathrm{t}_1 : \mathrm{Tree},\ \mathrm{t}_2 : \mathrm{Tree}}{[\mathrm{t}_1],\ \mathrm{t}_2 : \mathrm{Branch}} \tag{T7}$$

$$\frac{\mathrm{t}_1 : \mathrm{Tree},\ \mathrm{t}_2 : \mathrm{Branch}}{[\mathrm{t}_1],\ \mathrm{t}_2 : \mathrm{Branch}} \tag{T8}$$

$$\frac{\mathrm{t}_1 : \mathrm{Tree},\ \mathrm{t}_2 : \mathrm{Branch}}{\mathrm{t}_1,\ \mathrm{t}_2 : \mathrm{Tree}} \tag{T9}$$

均和evaluation规则有联系，意义很明确，不需要太多解释。

Safety

(Lemma of Canonical Form) 以下结论是显然的：

若t : Tree且t是value，则t = tree。

若t: Branch且t是value，则t = branch。

若t: Transform，则t = transform。

Progress

若t:T，则t是value，或者存在t'，t->t'。

对t: T的情形进行讨论。

(1) t=tree (3) t=branch (4)t=Transform t已经是value，平凡情况

(2) t=module，T=Tree。

根据(E3)，t->tree。

(5) t = t1, t2，T=Tree。且t1:Tree, t2:Tree

若t1是value，则t1=tree1。这时，若t2->t2'，则根据(E2)，t->tree1, t2'；若t2也是value，t2=tree2，则根据(E4)，t->tree'。

若t1->t1',根据(E1)，t->t1', t2

(6) t = t1, t2，T=Tree。且t1:Tree, t2:Transform

若t1是value，t1=tree1，且必t2 = transform2。根据(E5)，t->tree1。

若t1->t1'，根据(E1)， t->t1', t2

(7) t = [t1], t2，T=Branch。且t1: Tree, t2:Tree

若t1是value，t1=tree1，且[tree1]也是value。此时若t2也是value，t2=tree2，则根据(E7)，t->branch；若t2->t2'，则根据(E2)，t->[tree1], t2'

若t1->t1'，则根据(E1)及(E6)，t->[t1'], t2

(8) t = [t1], t2，T=Branch。 且t1: Tree, t2:Branch

若t1是value，t1=tree1，且[tree1]也是value。此时若t2也是value，则t2=branch，根据(E8)，t->branch'；若t2->t2'，则根据(E2)，t->[tree1], t2'。

若t1->t1'，则根据(E1)及(E6)，t->[t1'],t2。

(9) t = t1, t2，T=Tree。 且t1: Tree, t2:Branch

若t1是value，t1=tree1。此时若t2也是value，则t2=branch，根据(E9)，t->tree'；若t2->t2'，则根据(E2)，t->tree1, t2'。

若t1->t1'，则根据(E1)，t->t1',t2。

Preservation

若t : T且t -> t', 那么t' : T.

对t:T的情况进行讨论。

(1)(3)(4)为平凡情况。

(2) t=module，T=Tree。

根据(E3)，t'=tree，又根据(T5)，t':Tree。

(5) t = t1, t2，T=Tree。且t1:Tree, t2:Tree

若t1是value，则t1=tree1。这时，若t2->t2'，由归纳假设有t2':Tree。根据(E2)，t'=tree1,t2'，又根据(T5)，t':Tree；若t2也是value，t2=tree2，则根据(E4)，t'=tree'，故t':Tree。

若t1->t1'，t1':Tree。t'=t1', t2，根据(T5)，t':Tree。

(6) t = t1, t2，T=Tree。且t1:Tree, t2:Transform

必有t2 = transform2。

若t1是value，则t1=tree1。则根据(E5)，t'=tree'，根据(T1)，t':Tree。

t1不是value的情况同(5)。

(7) t = [t1], t2，T=Branch。且t1: Tree, t2:Tree

若t1是value，t1=tree1，且[tree1]也是value。此时若t2也是value，t2=tree2，则根据(E7)，t'=branch，根据(T3)，t':Branch；若t2->t2'，t2':Tree。则根据(E2)，t'=[tree1], t2'，又根据(T7)，t':Branch。

若t1->t1'，t1':Tree，则根据(E1)及(E6)，t'=[t1'], t2，又根据(T7)，t':Branch。

(8) t = [t1], t2，T=Branch。 且t1: Tree, t2:Branch

若t1是value，t1=tree1，且[tree1]也是value。此时若t2也是value，则t2=branch，根据(E8)，t'=branch'，又根据(T3)，t':Branch；若t2->t2'，t2':Branch，则根据(E2)，t'=[tree1], t2'，又根据(T8)，t':Branch。

若t1->t1'，t1':Tree，则根据(E1)及(E6)，t'=[t1'],t2，又根据(T8)，t':Branch。

(9) t = t1, t2，T=Tree。 且t1: Tree, t2:Branch

若t1是value，t1=tree1。此时若t2也是value，则t2=branch，根据(E9)，t'=tree'，又根据(T1)，t':Tree；若t2->t2'，t2':Tree，则根据(E2)，t'=tree1, t2'，又根据(T5)，t':Tree。

若t1->t1'，则根据(E1)，t'=t1',t2，又根据(T9)，t':Tree。

实现和应用

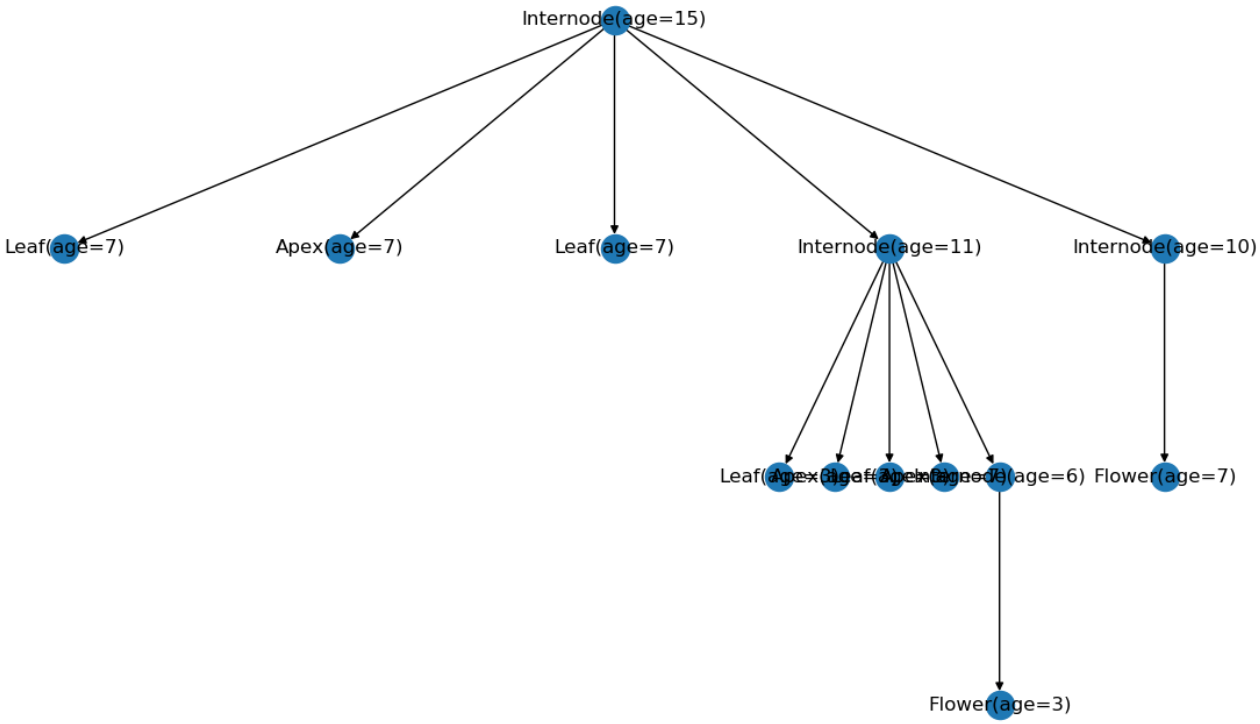
我们的L-system在Python中以面向对象的方式实现。其中的各种Module继承自Module类，L-string中存储的是Module类和Transform类的对象。图形可视化的部分借助了开源软件Blender[4]，我们的L-system描述文件可以作为其脚本执行，得到建模结果。下图是一张使用上下文相关产生式建模的结果图（mycelis），其规则设计的想法可以在[5]中找到。在这里，Module表示在Blender中预先建立好的模型，L-system实际上起到了模型部件"组装"的功能。

由于term的构成比较简单，不需要进行额外的语法解析，我们直接在L-string的nested list上进行evaluation和typing。仅通过使用分支语句，就可以覆盖evaluation和typing rules的情况。

通过evaluation，我们可以从Module和Transform的序列得到树拓扑，并通过networkx进行可视化。如下图，是L-string

> [Internode(age=15), [RotateY(50), Leaf(age=7)], RotateZ(90), [RotateX(-30), Apex(age=7)], RotateZ(90), [RotateY(50), Leaf(age=7)], RotateZ(90), [RotateX(-30), Internode(age=11), [RotateY(50), Leaf(age=3)], RotateZ(90), [RotateX(-30), Apex(age=3)], RotateZ(90), [RotateY(50), Leaf(age=3)], RotateZ(90), [RotateX(-30), Apex(age=7)], Internode(age=6), Flower(age=3)], Internode(age=10), Flower(age=7)]

所表示的树：



evaluation可以帮助快速了解L-string的结构，而直接去看string的话可读性不强，这对debug有一定帮助。typing则在上下文匹配前被调用，若L-string的类型为Tree，则保证这个string的表示是规范的。因此，把L-string看成是标记语言，其evaluation和typing作为系统的一部分，起到了重要的辅助功能。

参考文献

[1] D. S. Ebert and F. K. Musgrave. Texturing & modeling: a procedural approach. Morgan Kaufmann, 2003.

[2] A. Lindenmayer. “Mathematical models for cellular interactions in development I. Filaments with one-sided inputs”. Journal of theoretical biology, 1968, 18(3): 280–299.

[3] P. Prusinkiewicz, M. Hammel, J. Hanan et al. “L-systems: from the theory to visual models of plants”. In: Proceedings of the 2nd CSIRO Symposium on Computational Challenges in Life Sciences, 1996: 1–32.

[4] <https://www.blender.org/

[5] P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants. Springer Science&Business Media, 2012.