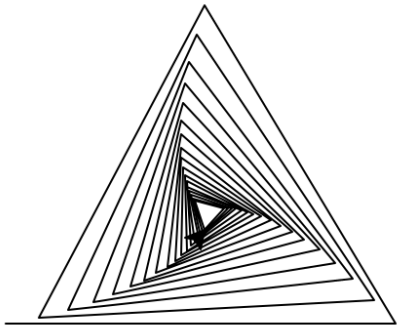
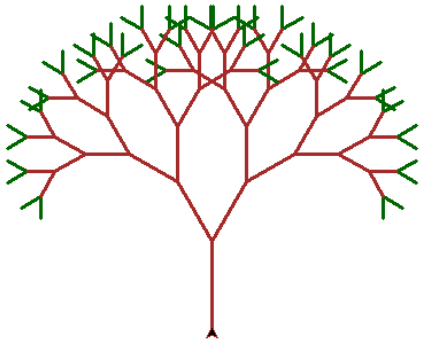
Turtle Recursion, Trees





CS111 Computer Programming

Department of Computer Science
Wellesley College

Review: Turtle Graphics

Concepts in this slide: A list of all useful functions from the turtle module.

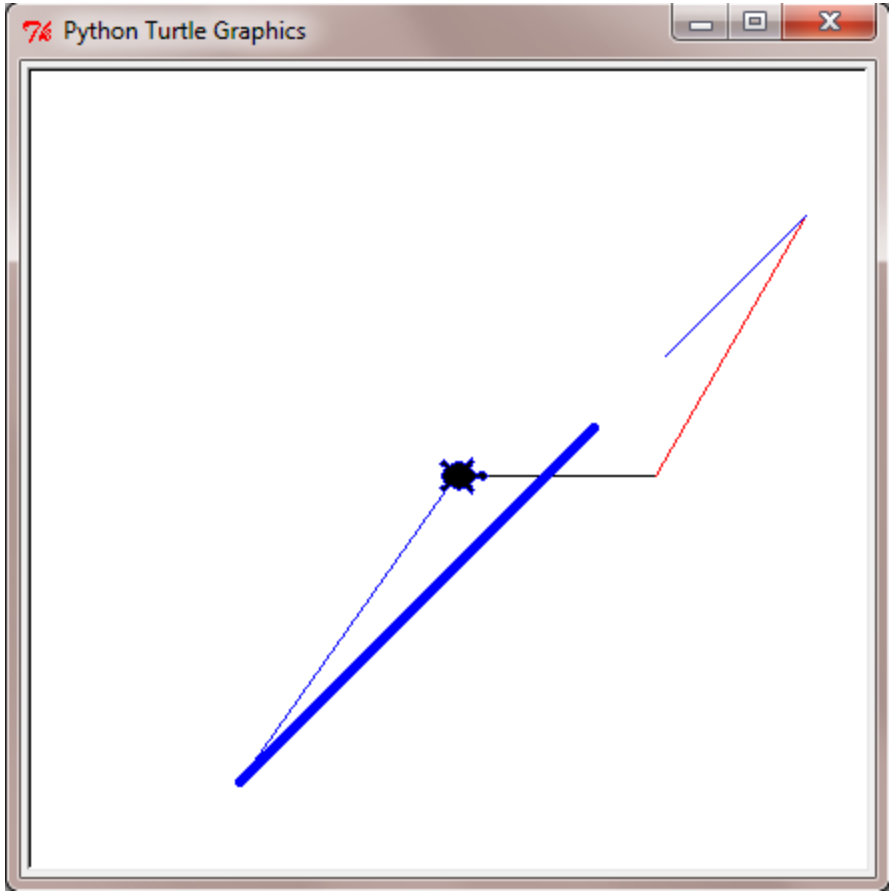
Python has a built-in module named **`turtle`**. See the Python turtle module API for details.

Use `from turtle import *` to use these commands:

`fd(dist)`	turtle moves forward by *dist*
`bk(dist)`	turtle moves backward by *dist*
`lt(angle)`	turtle turns left *angle* degrees
`rt(angle)`	turtle turns right *angle* degrees
`pu()`	(pen up) turtle raises pen in belly
`pd()`	(pen down) turtle lower pen in belly
`pensize(width)`	sets the thickness of turtle's pen to *width*
`pencolor(color)`	sets the color of turtle's pen to *color*
`shape(shp)`	sets the turtle's shape to *shp*
`home()`	turtle returns to (0,0) (center of screen)
`clear()`	delete turtle drawings; no change to turtle's state
`reset()`	delete turtle drawings; reset turtle's state
`setup(width,height)`	create a turtle window of given *width* and *height*

A Simple Example with Turtle

Concepts in this slide: The only two commands that draw lines are **fd** and **bk**.




```
from turtle import *

setup(400,400)
fd(100)
lt(60)
shape('turtle')
pencolor('red')
fd(150)
rt(15)
pencolor('blue')
bk(100)
pu()
bk(50)
pd()
pensize(5)
bk(250)
pensize(1)
home()
exitonclick()
```

Tk window

The turtle module has its own graphics environment that is created when we call the function **setup**. All drawing happens in it.

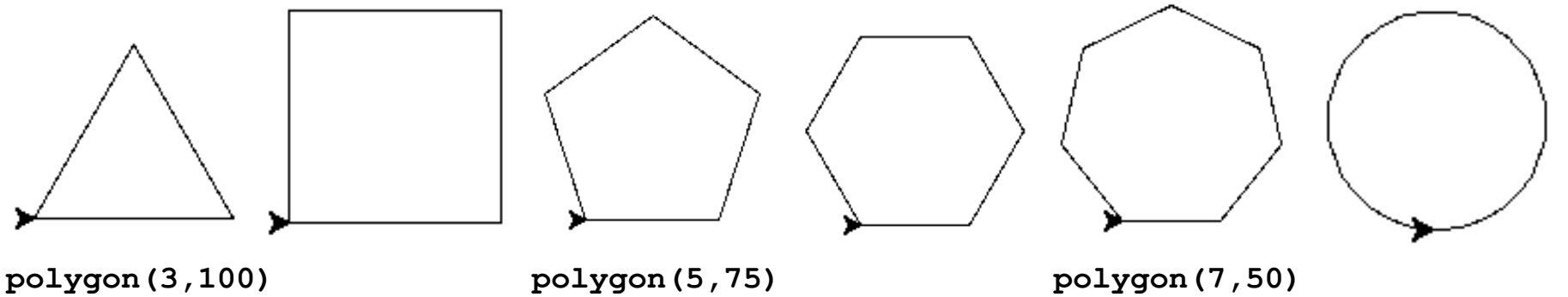
Looping Turtles (1)

Concepts in this slide: The power of abstraction: one function that creates a myriad of different shapes.

Loops can be used in conjunction with turtles to make interesting designs.

```
def polygon(numSides, sideLength):
    """ Draws a polygon with the specified number
    of sides, each with the specified length.
    """
```

Will solve this in the Notebook.

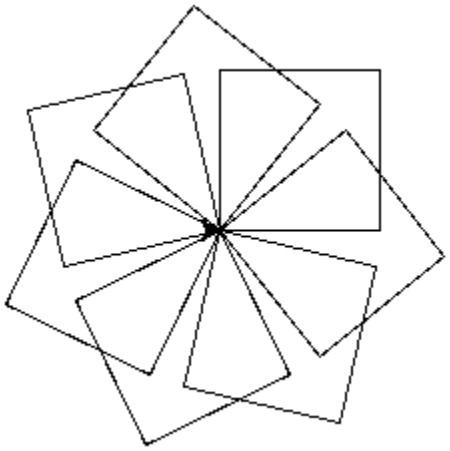


Looping Turtles (2)

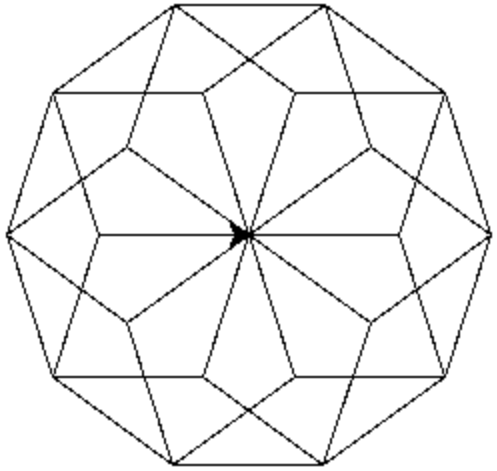
```
def polyFlow(numPetals, petalSides, petalLen):
    """Draws 'flowers' with numPetals arranged around
    a center point. Each petal is a polygon with
    petalSides sides of length petalLen.
    """
```

Will solve this in the Notebook.

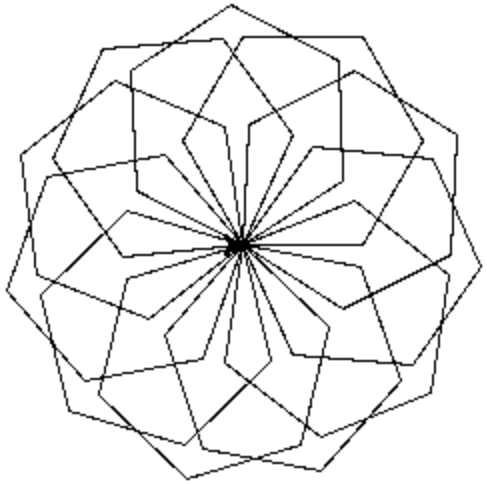
polyFlow(7,4,80)



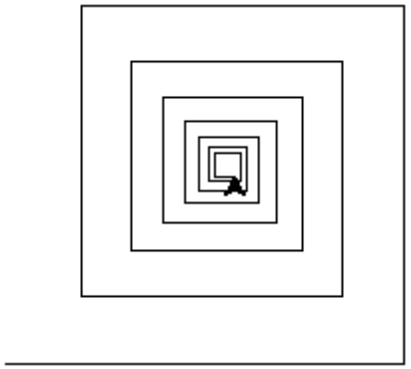
polyFlow(10,5,75)



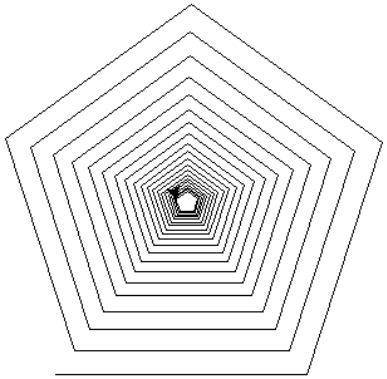
polyFlow(11,6,60)



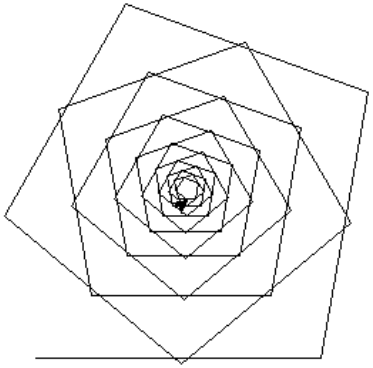
Spiraling Turtles: A Recursion Example



`spiral(200,90,0.9,10)`



`spiral(200,72,0.97,10)`

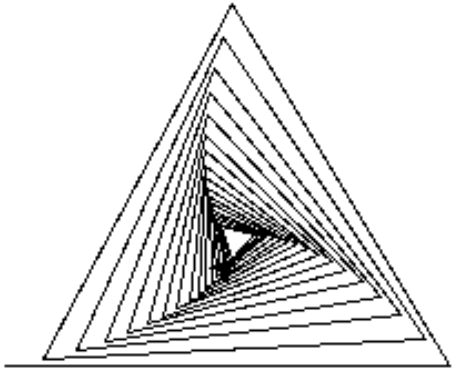


`spiral(200,80,0.95,10)`

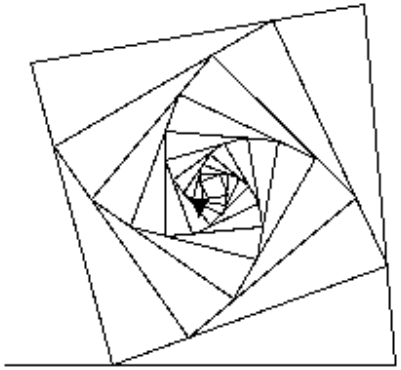
Answer this:

How would you create these shapes using loops?

Recursion makes easier solving certain problems that involve a repeating pattern.



`spiral(200,121,0.95,15)`



`spiral(200,95,0.93,10)`

Reminder: Structure of Recursion

All recursive functions must have two types of cases:

- **BASE case**: a simple case where the solution is obvious.
  In this case the function does **not** invoke itself, since there is no need to decompose the problem into subproblems. *Sometimes, we can leave out the base case, if it doesn't do anything.*

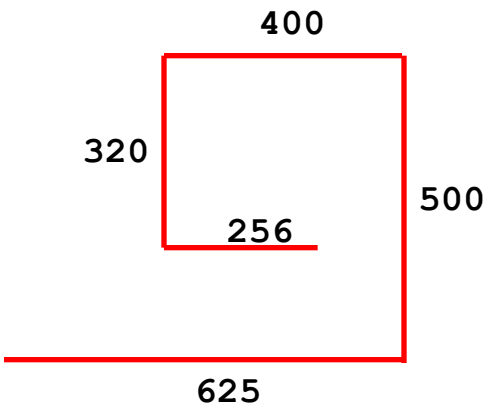
- **RECURSIVE case**: a case where the problem
  - is decomposed into subproblems
  - at least one of the subproblems is solved by **invoking the function being defined**, i.e., the function is invoked in its own body. You should assume the recursive function works correctly for **the smaller subproblems** (this is known as "wishful thinking")

Spiraling Turtles: A Recursion Example

```
def spiral(sideLen, angle,
           scaleFactor, minLength):
    """Draw a spiral recursively."""

    if sideLen >= minLength:
        fd(sideLen)
        lt(angle)
        spiral(sideLen*scaleFactor,
               angle,
               scaleFactor,
               minLength)
```

- **sideLen** is the length of the current side
- **angle** is the amount the turtle turns left to draw the next side
- **scaleFactor** is the multiplicative factor (between 0.0 and 1.0) by which to scale the next side
- **minLength** is the smallest side length that the turtle will draw




```
spiral(625, 90, 0.8, 250)
```

Concepts in this slide: Drawing function call frames helps us follow the execution of recursion.

```
spiral(625, 90, 0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
if sideLen >= minLength:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
if True:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```


```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```


```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```


```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

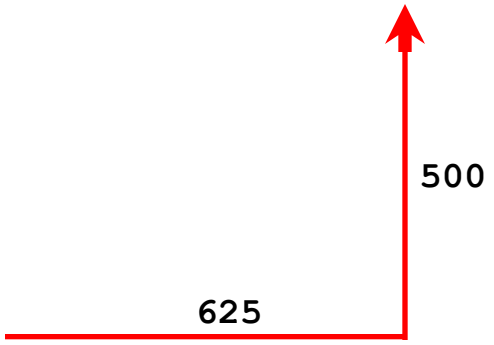
```
spiral(500, 90, 0.8, 250)
if sideLen >= minLength:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```


```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

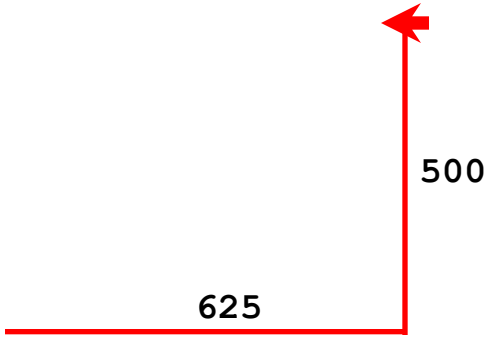
```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

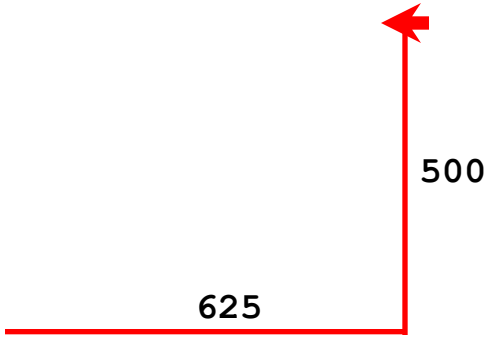
```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

spiral(625, 90, 0.8, 250)

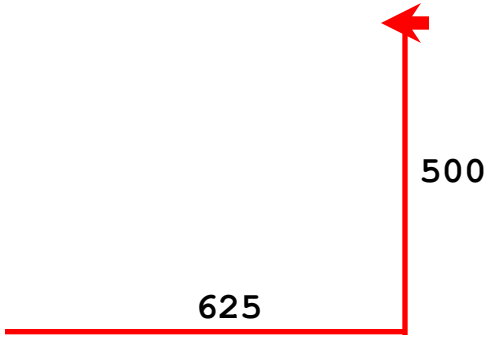



```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if sideLen >= minLength:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

spiral(625, 90, 0.8, 250)

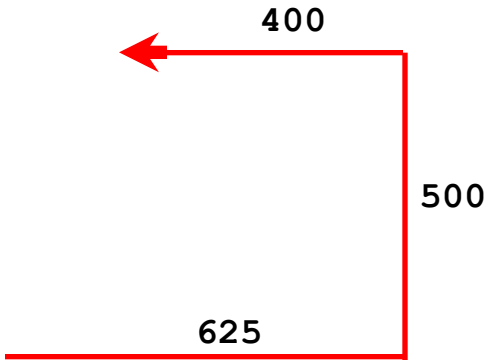



```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```

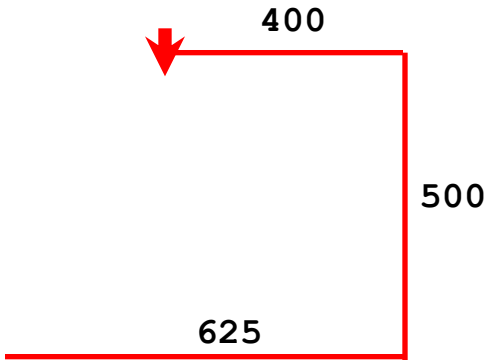



```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```

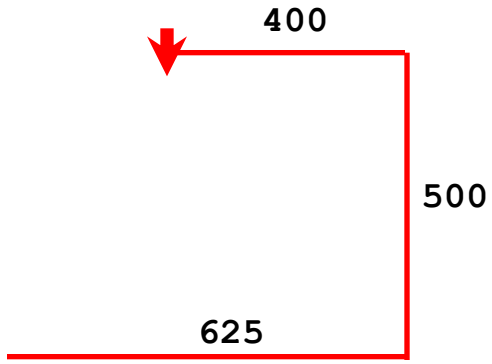



```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```



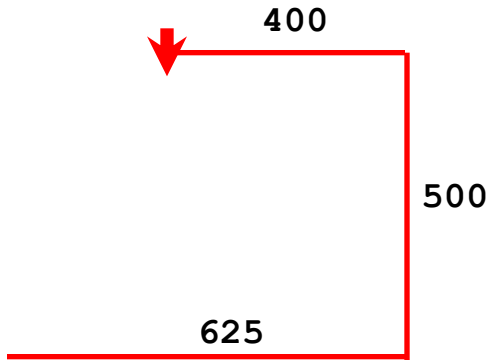

```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if sideLen >= minLength:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```



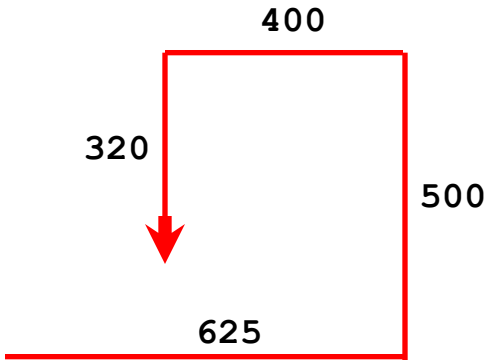

```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```



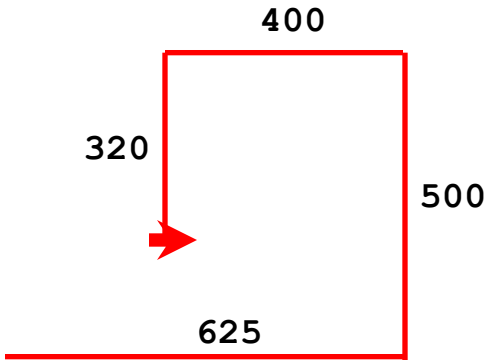

```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```



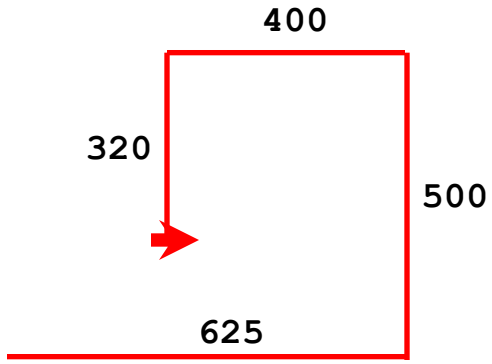

```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

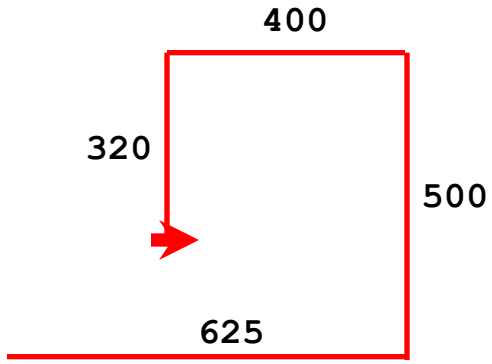
```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(256, 90, 0.8, 250)
if sideLen >= minLength:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

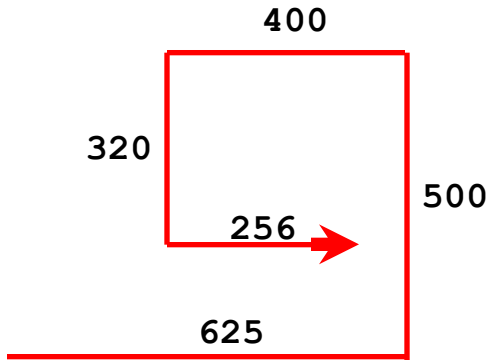
```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(256, 90, 0.8, 250)
if True:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

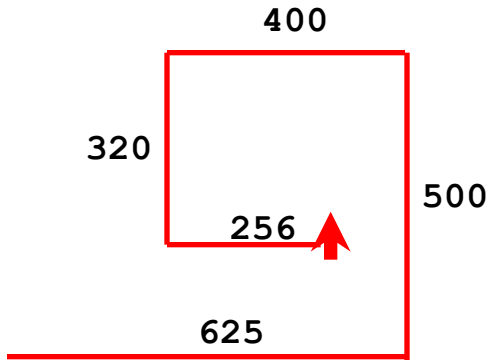
```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(256, 90, 0.8, 250)
if True:
  fd(256)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

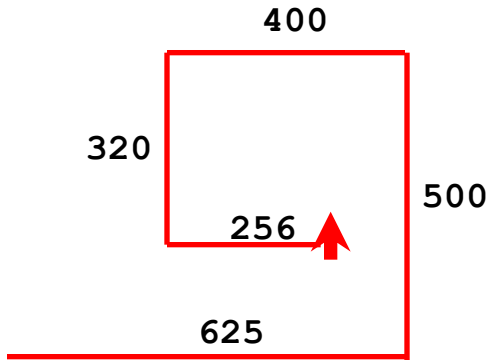
```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(256, 90, 0.8, 250)
if True:
  fd(256)
  lt(90)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

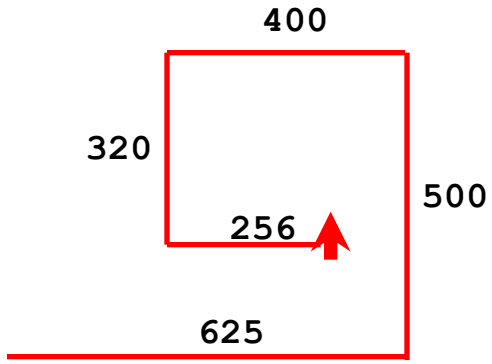
```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(204.8, 90, 0.8, 250)
if sideLen >= minLength:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(256, 90, 0.8, 250)
if True:
  fd(256)
  lt(90)
  spiral(204.8, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

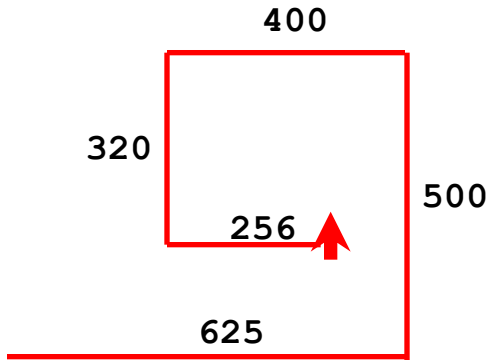
```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(256, 90, 0.8, 250)
if True:
  fd(256)
  lt(90)
  spiral(204.8, 90,
         0.8, 250)
```

```
spiral(204.8, 90, 0.8, 250)
if False:
  fd(sideLen)
  lt(angle)
  spiral(sideLen*scaleFactor, angle,
         scaleFactor, minLength)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

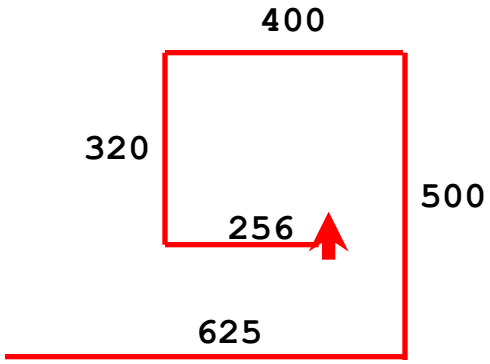
```
spiral(256, 90, 0.8, 250)
if True:
  fd(256)
  lt(90)
  spiral(204.8, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

Important

Initially all execution frames co-exist in the memory. Only once a function has returned (implicitly), the execution frame is deleted.

```
spiral(625, 90, 0.8, 250)
```



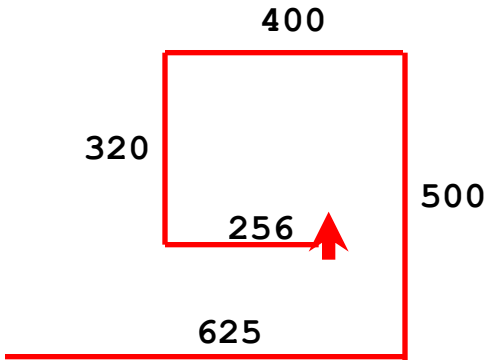

```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

```
spiral(320, 90, 0.8, 250)
if True:
  fd(320)
  lt(90)
  spiral(256, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```

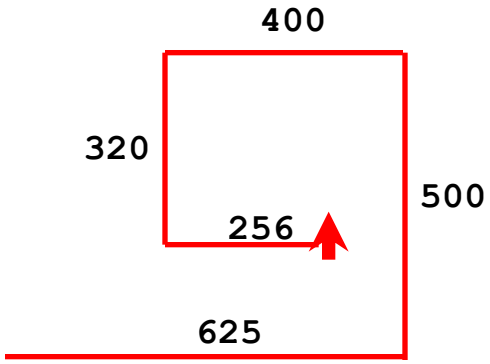



```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(400, 90, 0.8, 250)
if True:
  fd(400)
  lt(90)
  spiral(320, 90,
         0.8, 250)
```

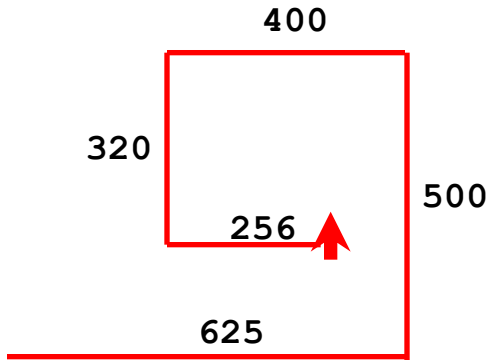
```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

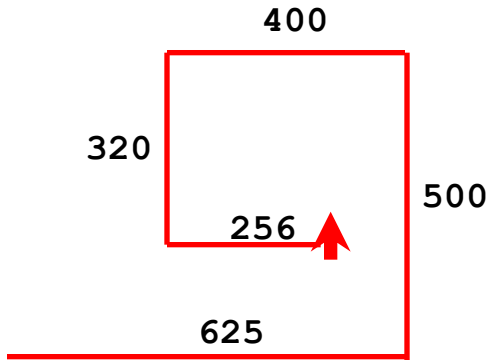
```
spiral(500, 90, 0.8, 250)
if True:
  fd(500)
  lt(90)
  spiral(400, 90,
         0.8, 250)
```

```
spiral(625, 90, 0.8, 250)
```




```
spiral(625, 90, 0.8, 250)
if True:
  fd(625)
  lt(90)
  spiral(500, 90,
         0.8, 250)
```

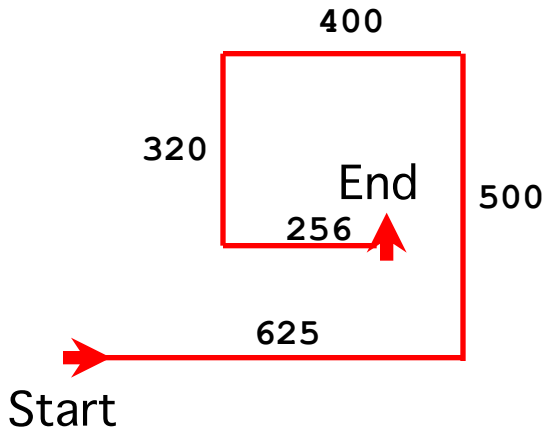
```
spiral(625, 90, 0.8, 250)
```



Important

All execution frames were one by one deleted after their completion. This terminates the invocation of the function and has created as a "side-effect" the turtle image at the top of the slide.

Problem: Where is the turtle?

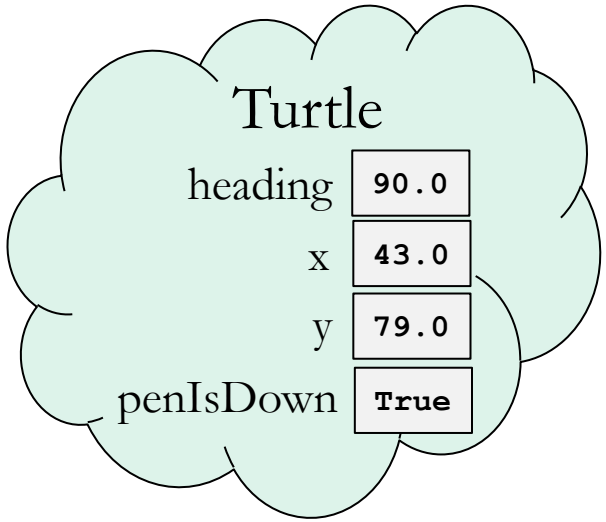


To notice

The turtle started in one position on the canvas and ended in another one (see Start and End labels in the drawing). That is not desirable, we want the turtle to be back to its original position on its own.

Invariant Spiraling

A function is invariant relative to an object's state if the state of the object is the same before and after the function is invoked.



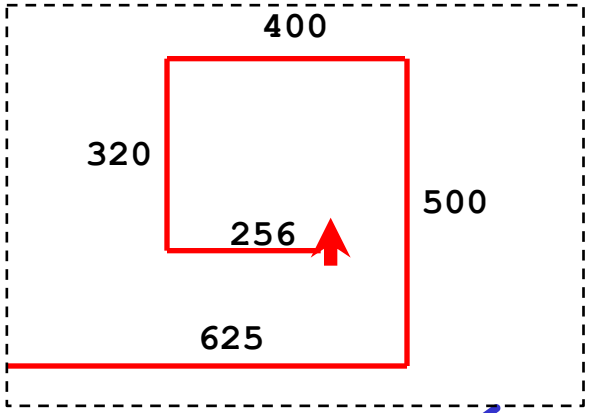
Memory diagram for turtle

A Turtle object resembles a dictionary, it has named properties that store their values, as the picture shows.

```
def spiralBack(sideLen, angle, scaleFactor, minLength):
    """ Draws a spiral. The state of the turtle
    (position, color, heading, etc.) after drawing
    the spiral is the same as before drawing the spiral.
    """
```

How does spiralBack work?

```
spiralBack(625, 90, 0.8, 250)
```



Be turtle, draw.

spiralBack(625, 90, 0.8, 250)

```
if True:                    sideLen 625   angle 90
  fd(625)
  lt(90)
  spiralBack(500, 90, 0.8, 250)
  rt(angle)
  bk(sideLen)
```

spiralBack(500, 90, 0.8, 250)

```
if True:                    sideLen 500   angle 90
  fd(500)
  lt(90)
  spiralBack(400, 90, 0.8, 250)
  rt(angle)
  bk(sideLen)
```

spiralBack(400, 90, 0.8, 250)

```
if True:                    sideLen 400   angle 90
  fd(400)
  lt(90)
  spiralBack(320, 90, 0.8, 250)
  rt(angle)
  bk(sideLen)
```

spiralBack(204.8, 90, 0.8, 250)

```
if False:                   sideLen 204.8 angle 90
  fd(sideLen)
  lt(angle)
  spiralBack(sideLen*scaleFactor,…)
  rt(angle)
  bk(sideLen)
```

spiralBack(256, 90, 0.8, 250)

```
if True:                    sideLen 256   angle 90
  fd(256)
  lt(90)
  spiralBack(204.8, 90, 0.8, 250)
  rt(angle)
  bk(sideLen)
```

spiralBack(320, 90, 0.8, 250)

```
if True:                    sideLen 320   angle 90
  fd(320)
  lt(90)
  spiralBack(256, 90, 0.8, 250)
  rt(angle)
  bk(sideLen)
```

Essence of Invariance

```
Do state change 1
Do state change 2
…
Do state change n-1
Do state change n
```

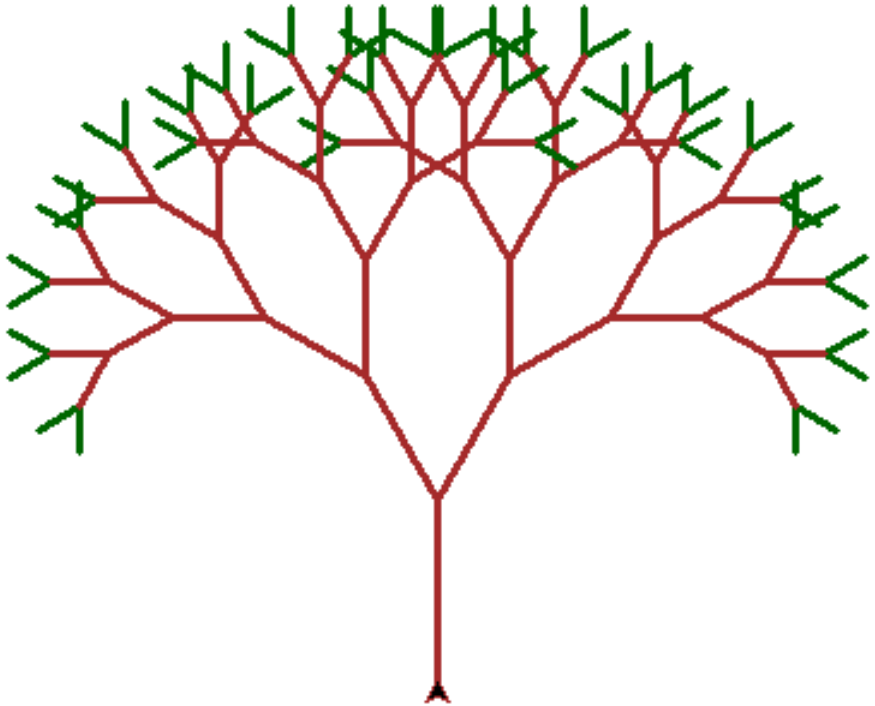
Perform changes to state

```
Recursive call to function
```

```
Undo state change n
Undo state change n-1
…
Undo state change 2
Undo state change 1
```

Undo state changes in opposite order

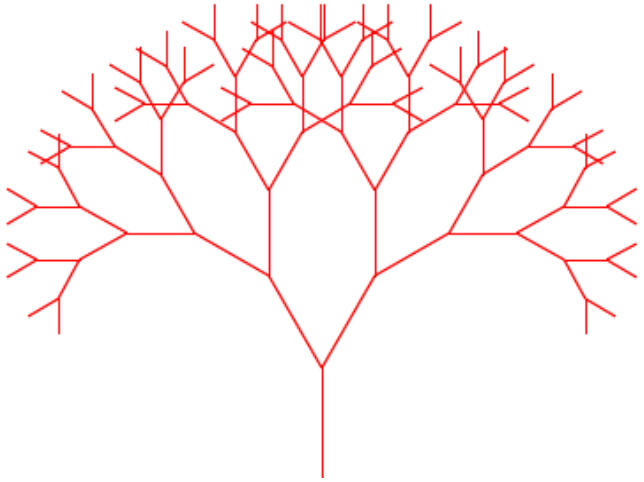
Drawing Trees with Recursion



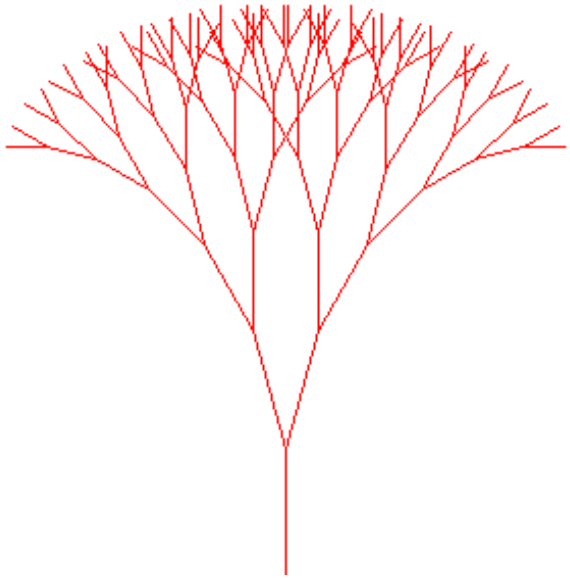
To notice

In this drawing, the turtle is back in its original position. That is, the tree was drawn by an invariant function.

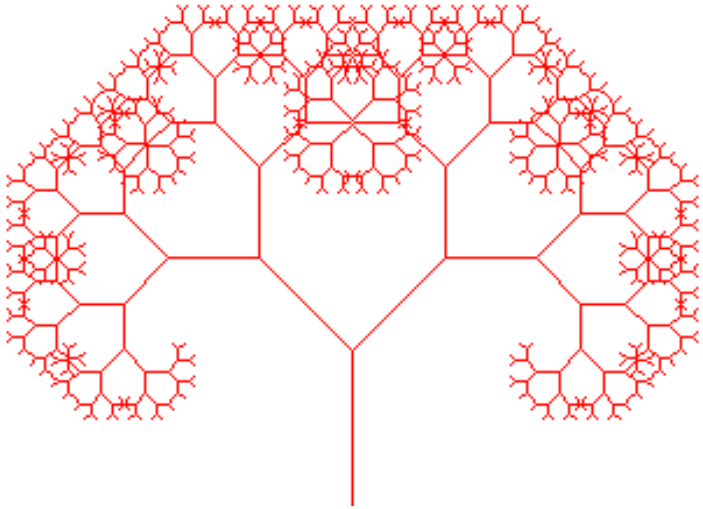
Trees



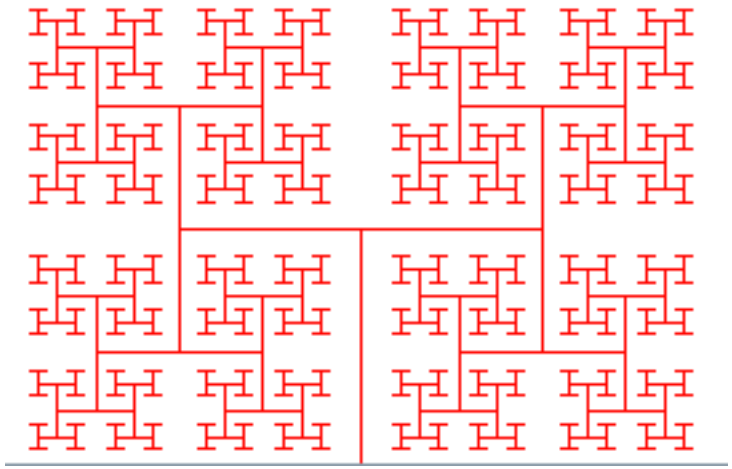
`tree(7, 75, 30, 0.8)`



`tree(7, 75, 15, 0.8)`



`tree(10, 80, 45, 0.7)`



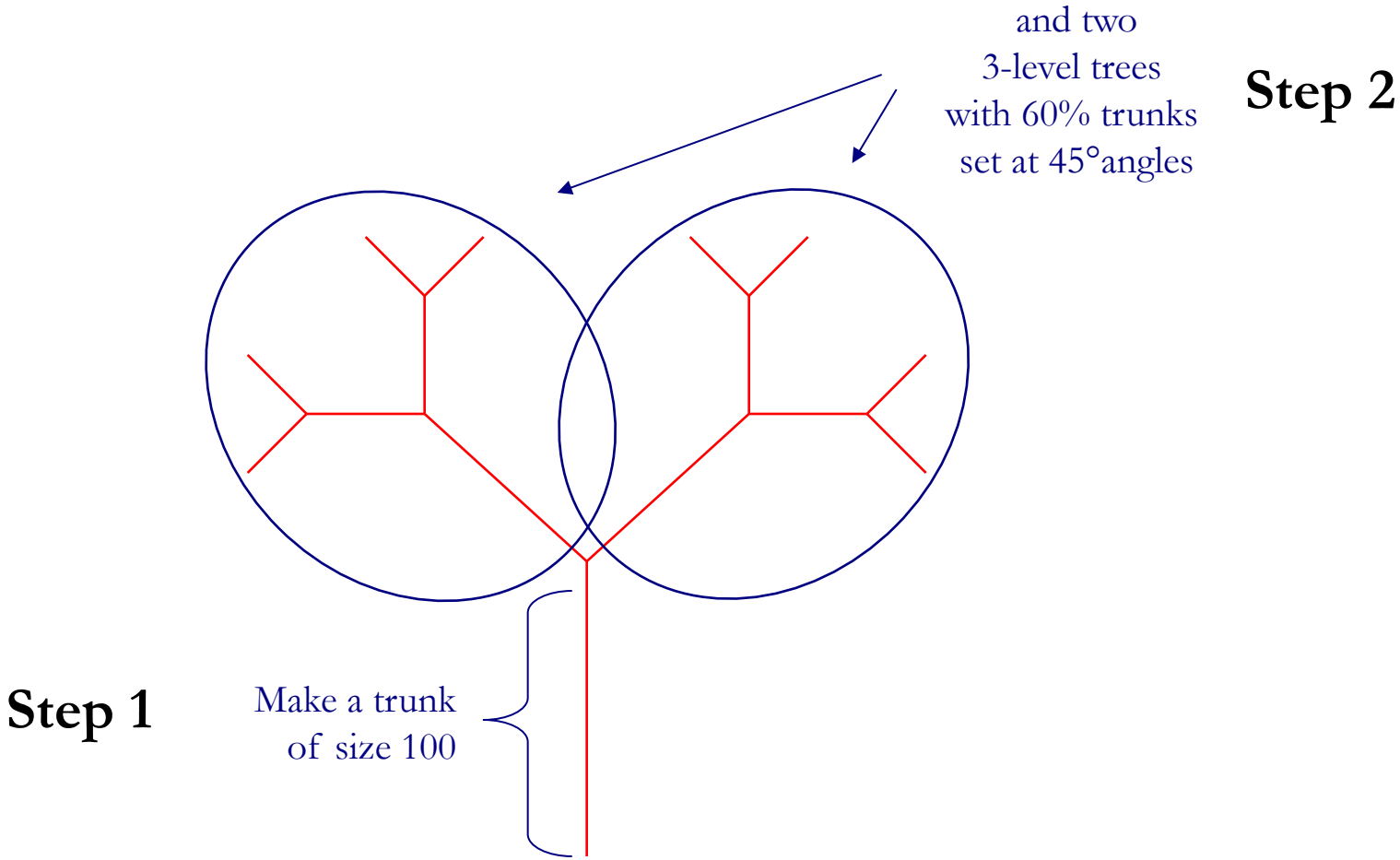
`tree(10, 100, 90, 0.68)`

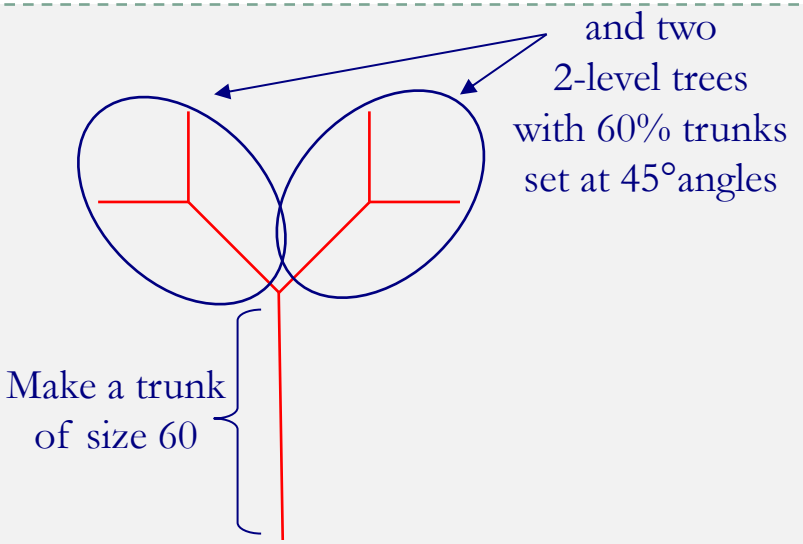
Draw a tree recursively

```
tree(levels, trunkLen, angle, shrinkFactor)
```

- **`levels`** is the number of branches on any path from the root to a leaf
- **`trunkLen`** is the length of the base trunk of the tree
- **`angle`** is the angle from the trunk for each subtree
- **`shrinkFactor`** is the shrinking factor for each subtree

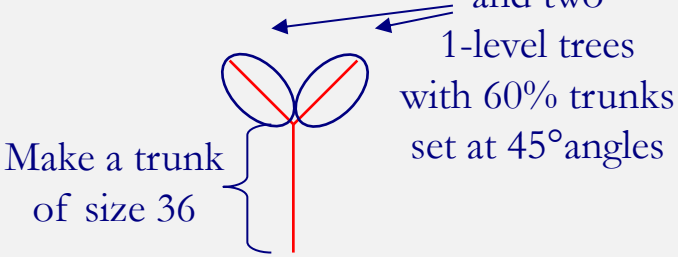
How to make a 4-level tree: `tree(4, 100, 45, 0.6)`





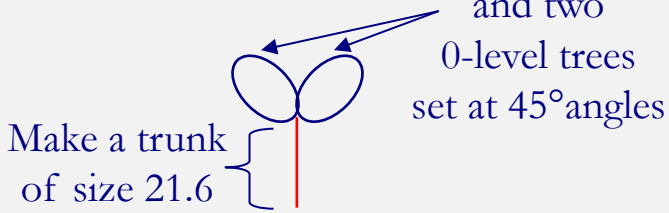
How to make a **3-level tree:**

```
tree(3, 60, 45, 0.6)
```



How to make a **2-level tree:**

```
tree(2, 36, 45, 0.6)
```



How to make a **1-level tree:**

```
tree(1, 21.6, 45, 0.6)
```

Do nothing!

How to make a **0-level tree:**

```
tree(0, 12.96, 45, 0.6)
```

```
def tree(levels, trunkLen, angle, shrinkFactor):
    """Draw a 2-branch tree recursively.

    levels: number of branches on any path
            from the root to a leaf
    trunkLen: length of the base trunk of the tree
    angle: angle from the trunk for each subtree
    shrinkFactor: shrinking factor for each subtree
    """
    if levels > 0:
        # Draw the trunk.
        fd(trunkLen)
        # Turn and draw the right subtree.
        rt(angle)
        tree(levels-1, trunkLen*shrinkFactor, angle, shrinkFactor)
        # Turn and draw the left subtree.
        lt(angle * 2)
        tree(levels-1, trunkLen*shrinkFactor, angle, shrinkFactor)
        # Turn back and back up to root without drawing.
        rt(angle)
        pu()
        bk(trunkLen)
        pd()
```

Tracing the invocation of `tree(3,60,45,0.6)`

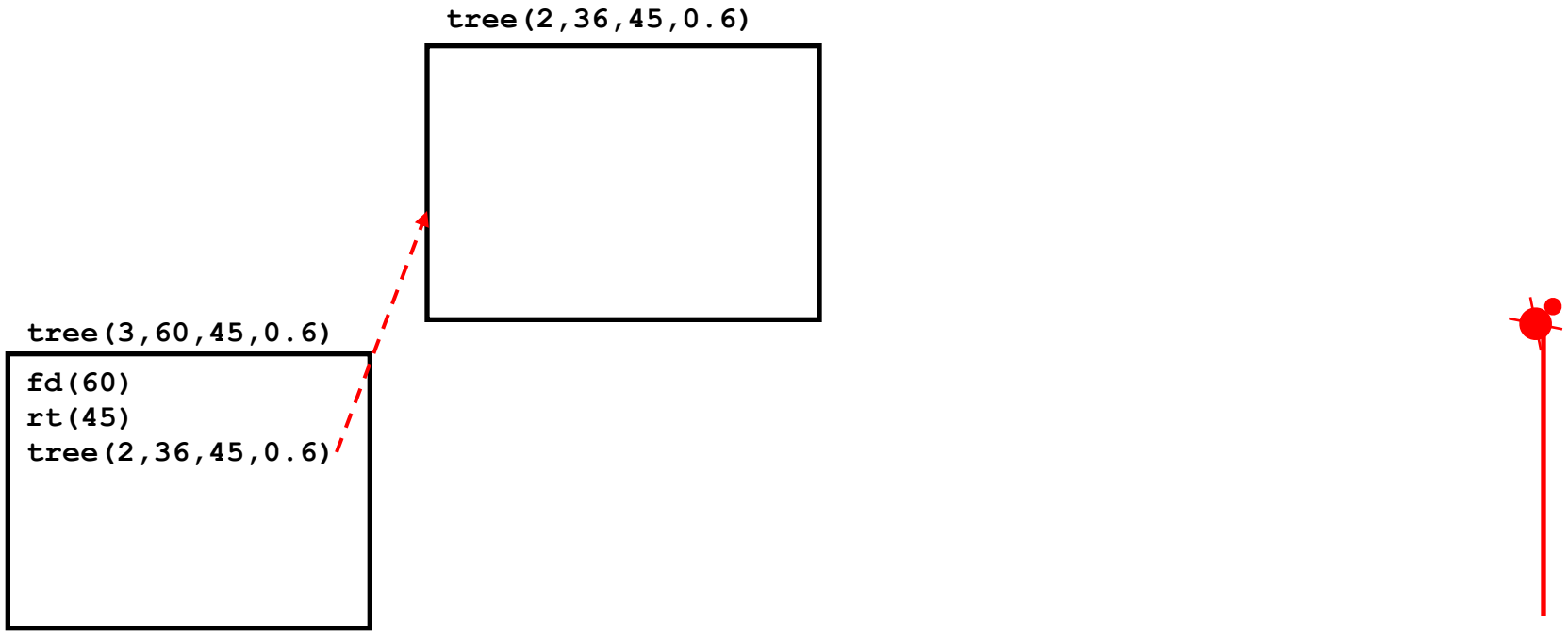
`tree(3,60,45,0.6)`

Draw trunk and turn to draw level 2 tree

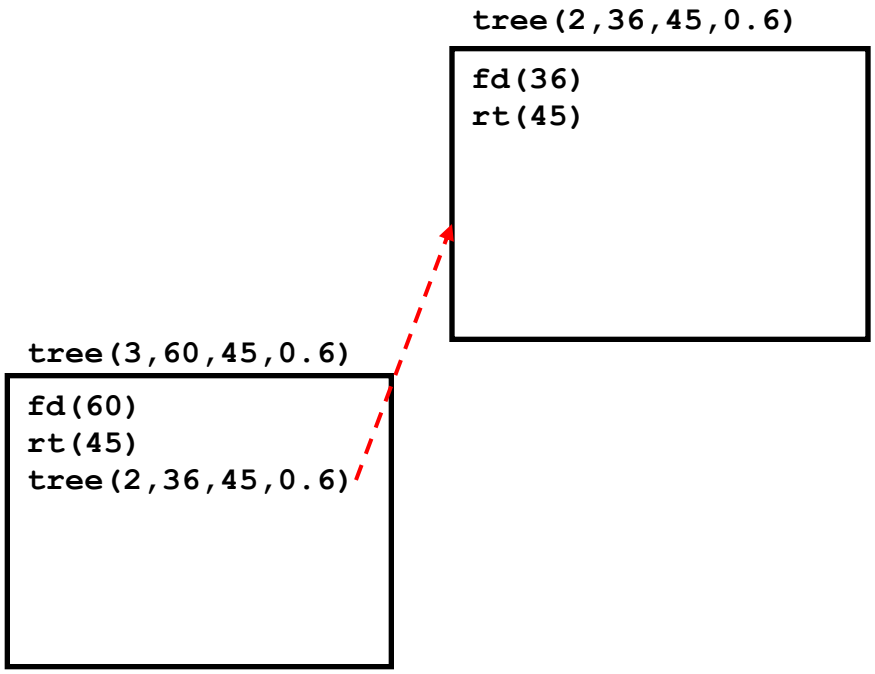
```
tree(3,60,45,0.6)
```

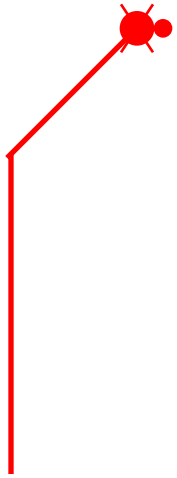
```
fd(60)
rt(45)
```

Begin recursive invocation to draw level 2 tree

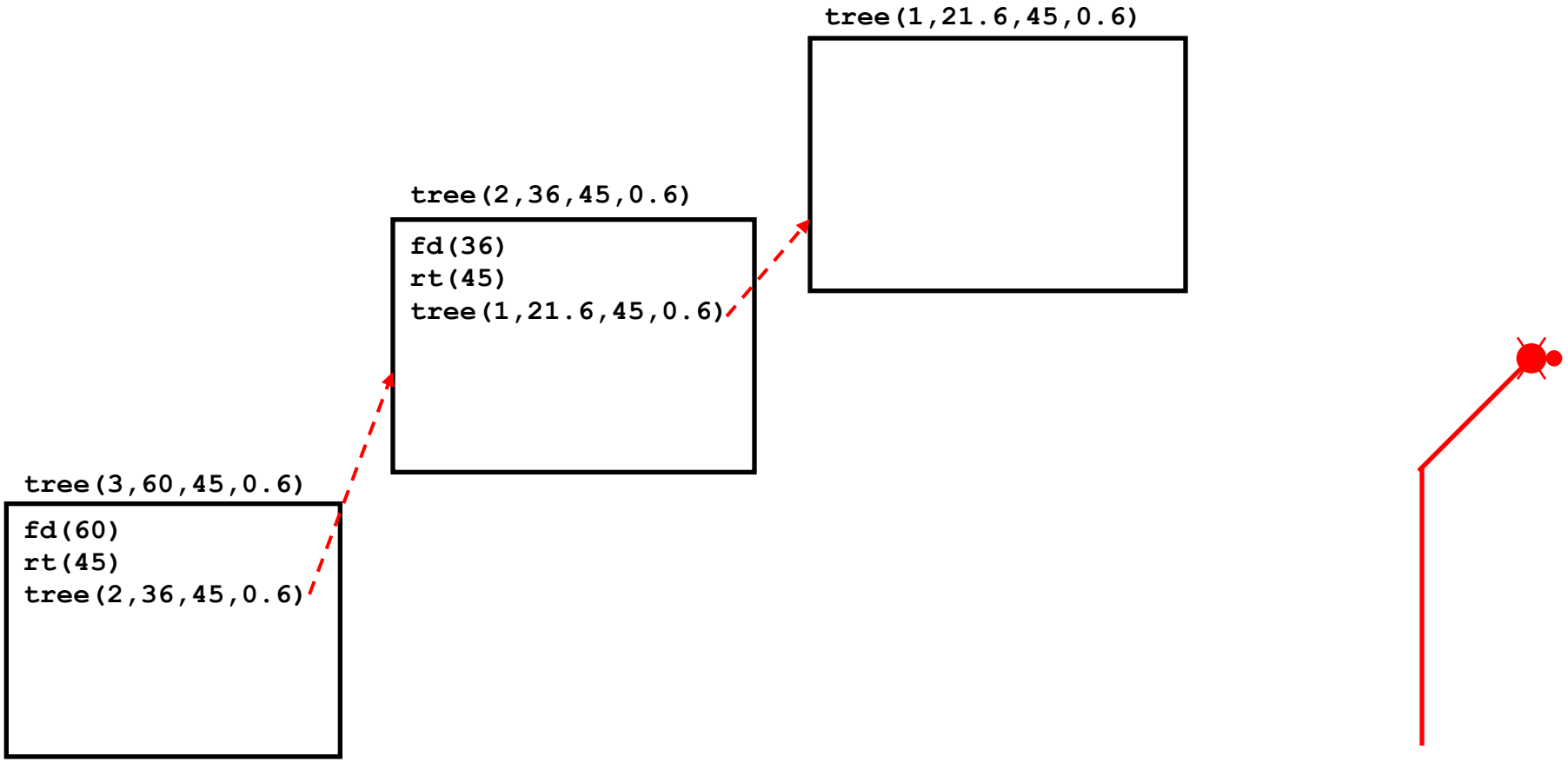


Draw trunk and turn to draw level 1 tree





Begin recursive invocation to draw level 1 tree

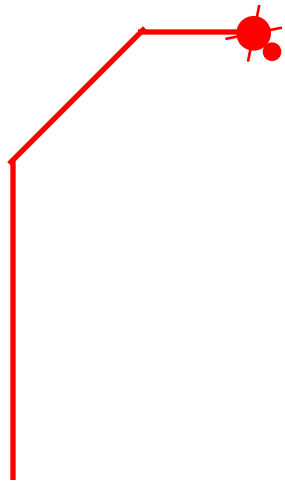


Draw trunk and turn to draw level 0 tree

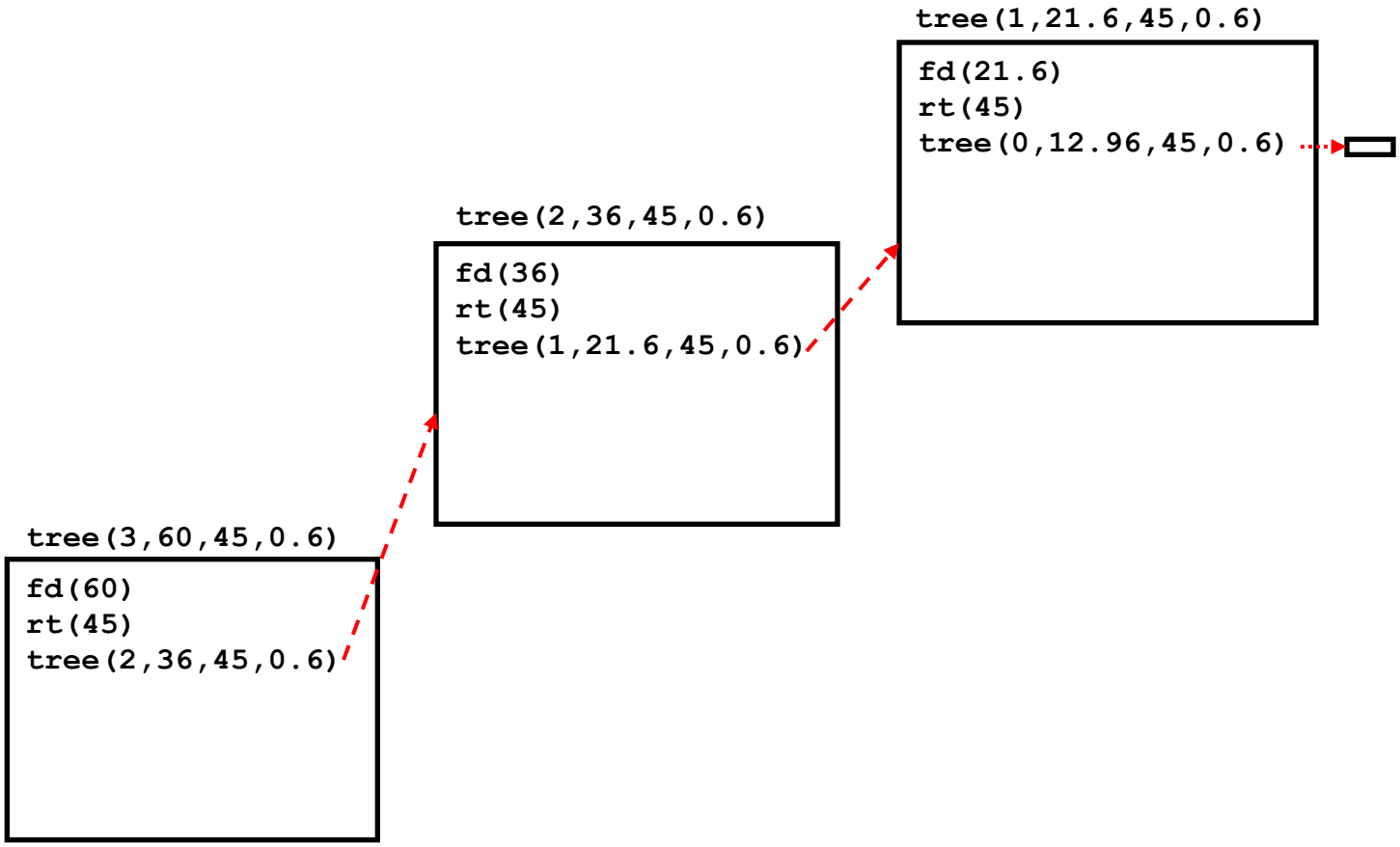
```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
```

```
tree(2,36,45,0.6)
fd(36)
rt(45)
tree(1,21.6,45,0.6)
```

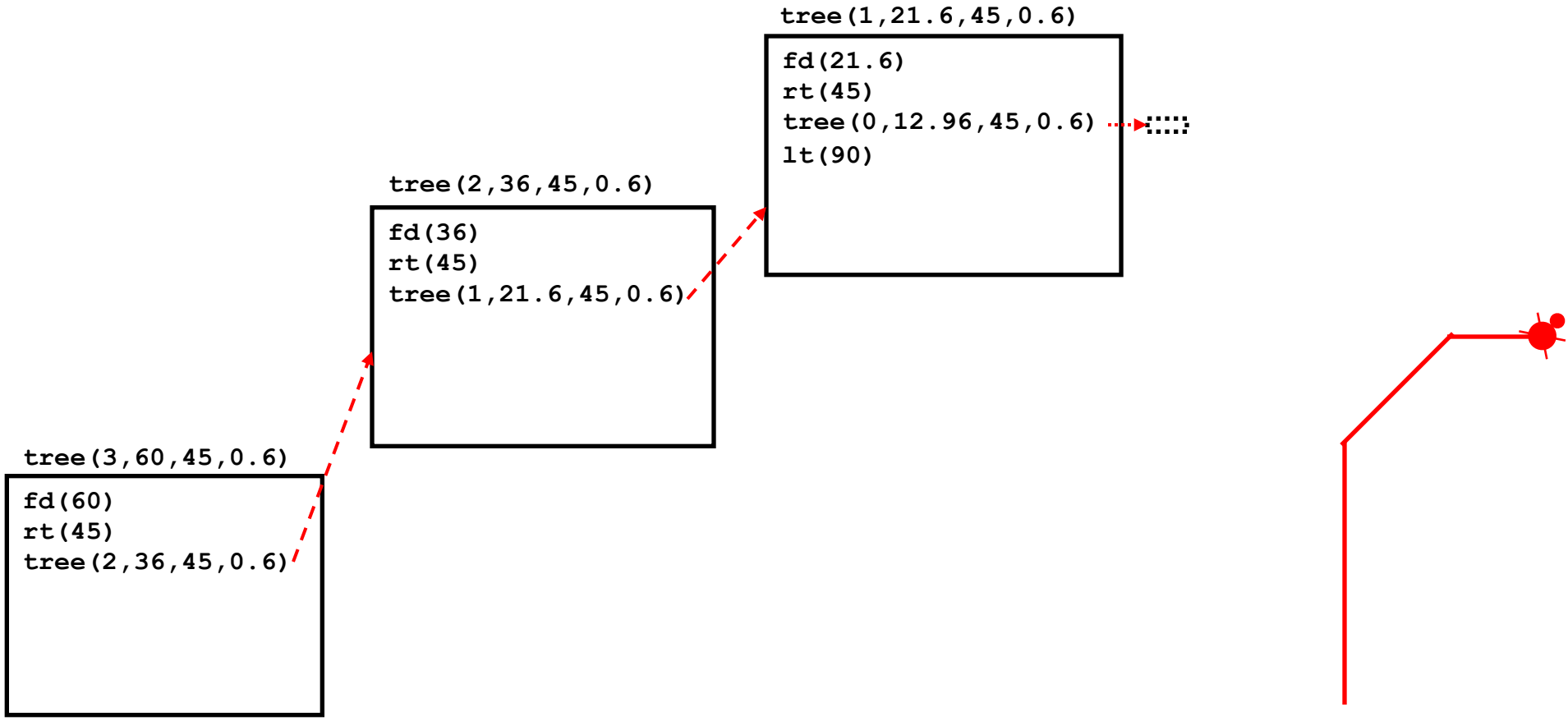
```
tree(3,60,45,0.6)
fd(60)
rt(45)
tree(2,36,45,0.6)
```



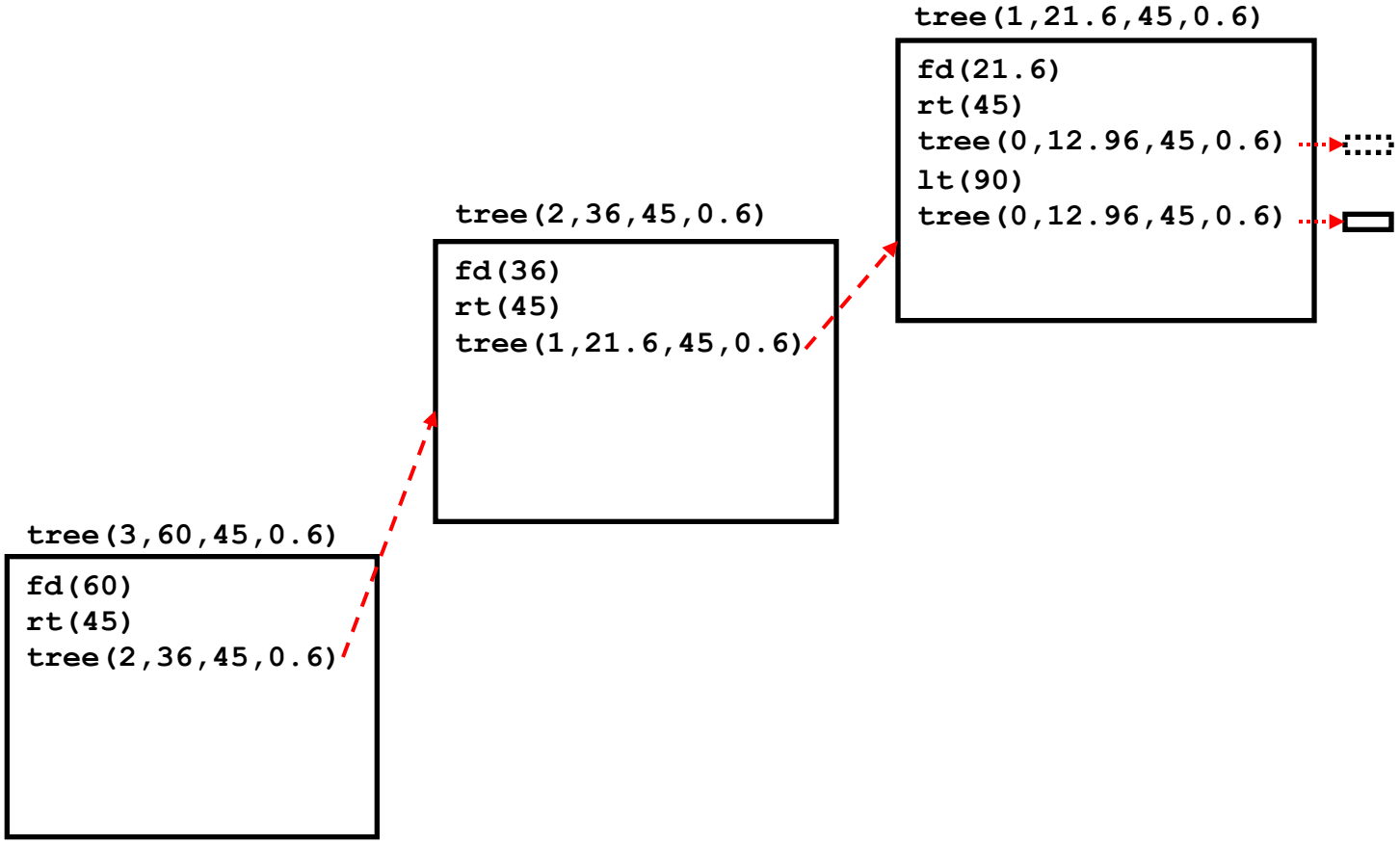
Begin recursive invocation to draw level 0 tree

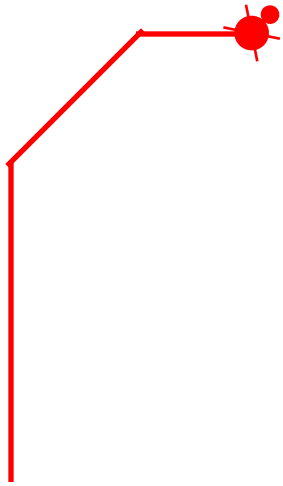


Complete level 0 tree and turn to draw another level 0 tree

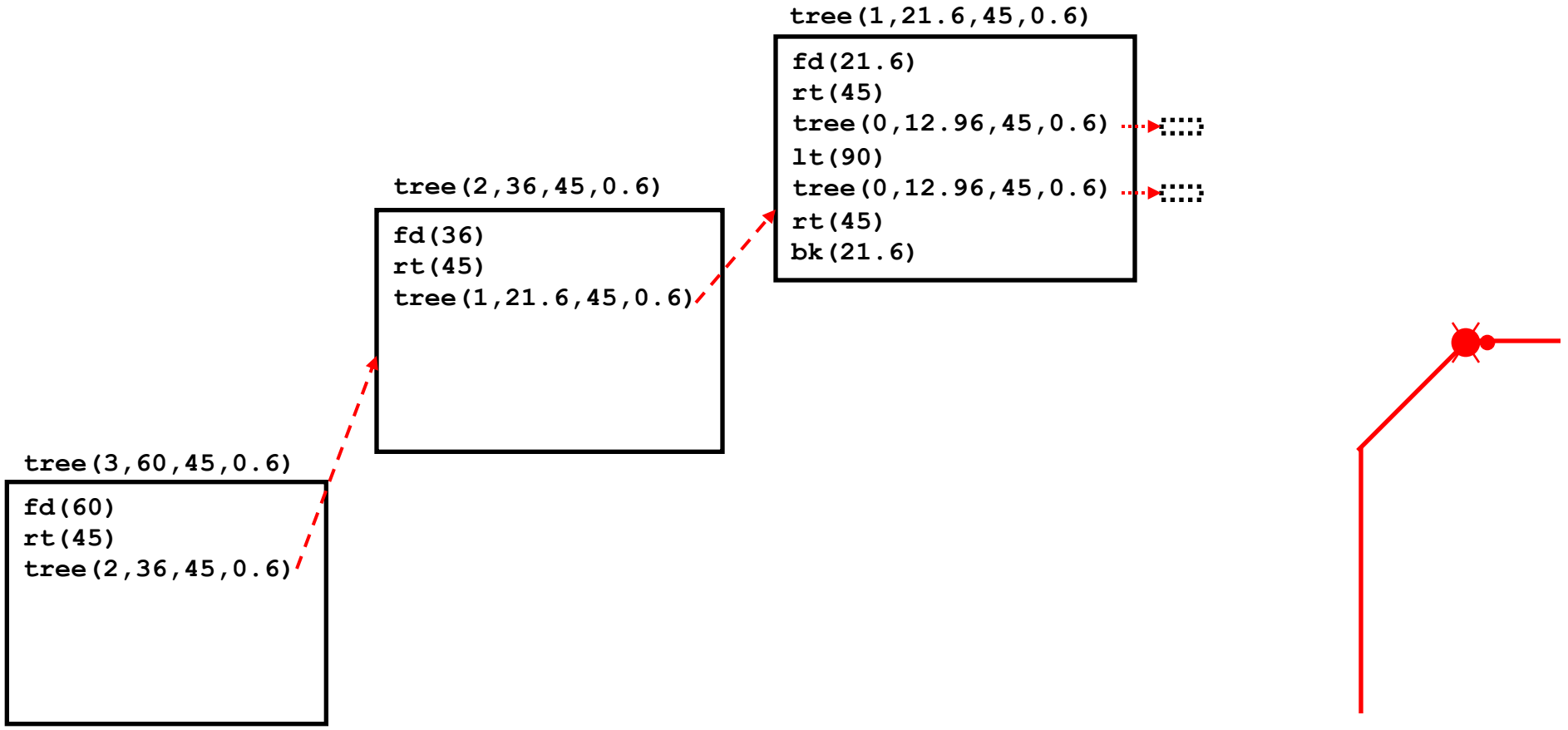


Begin recursive invocation to draw level 0 tree

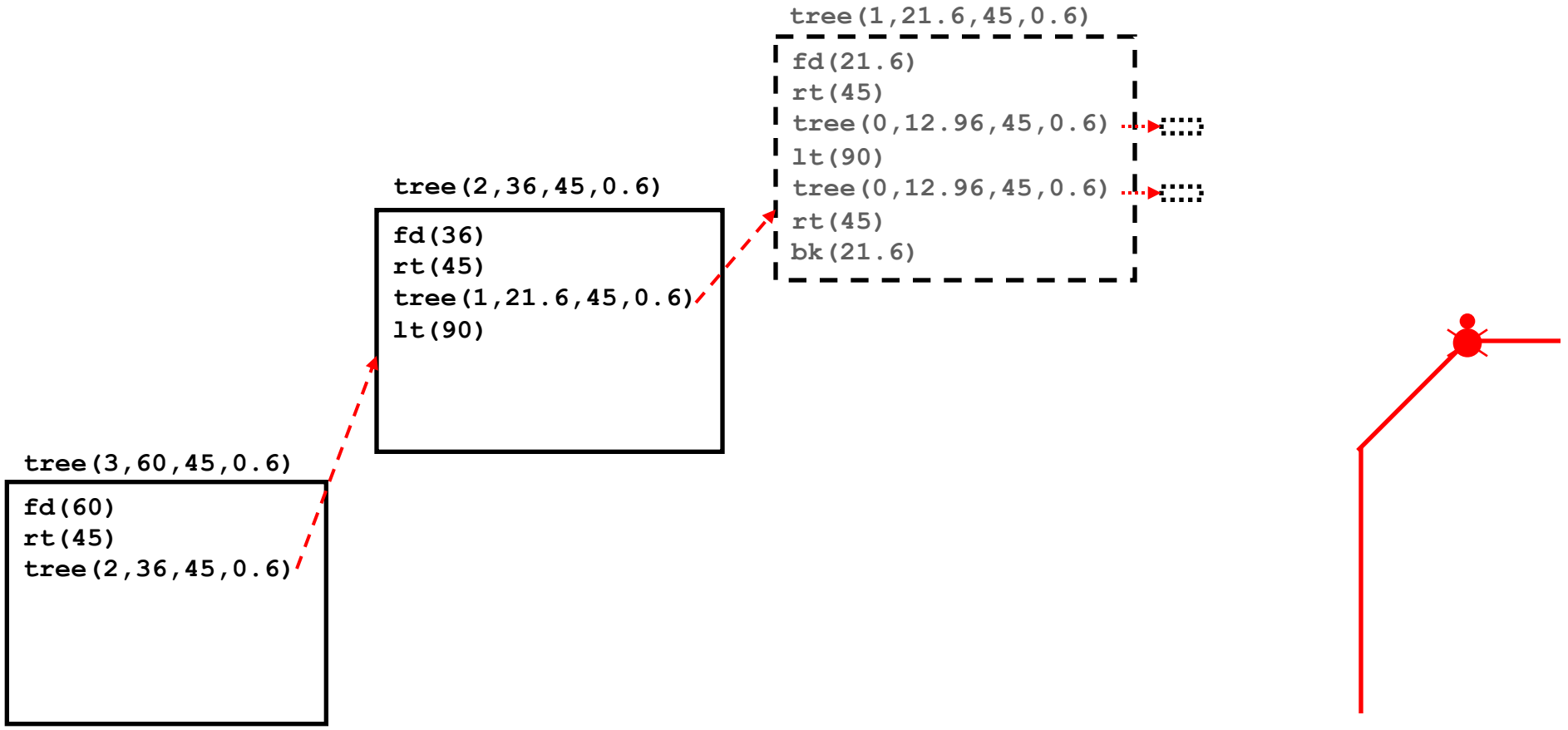




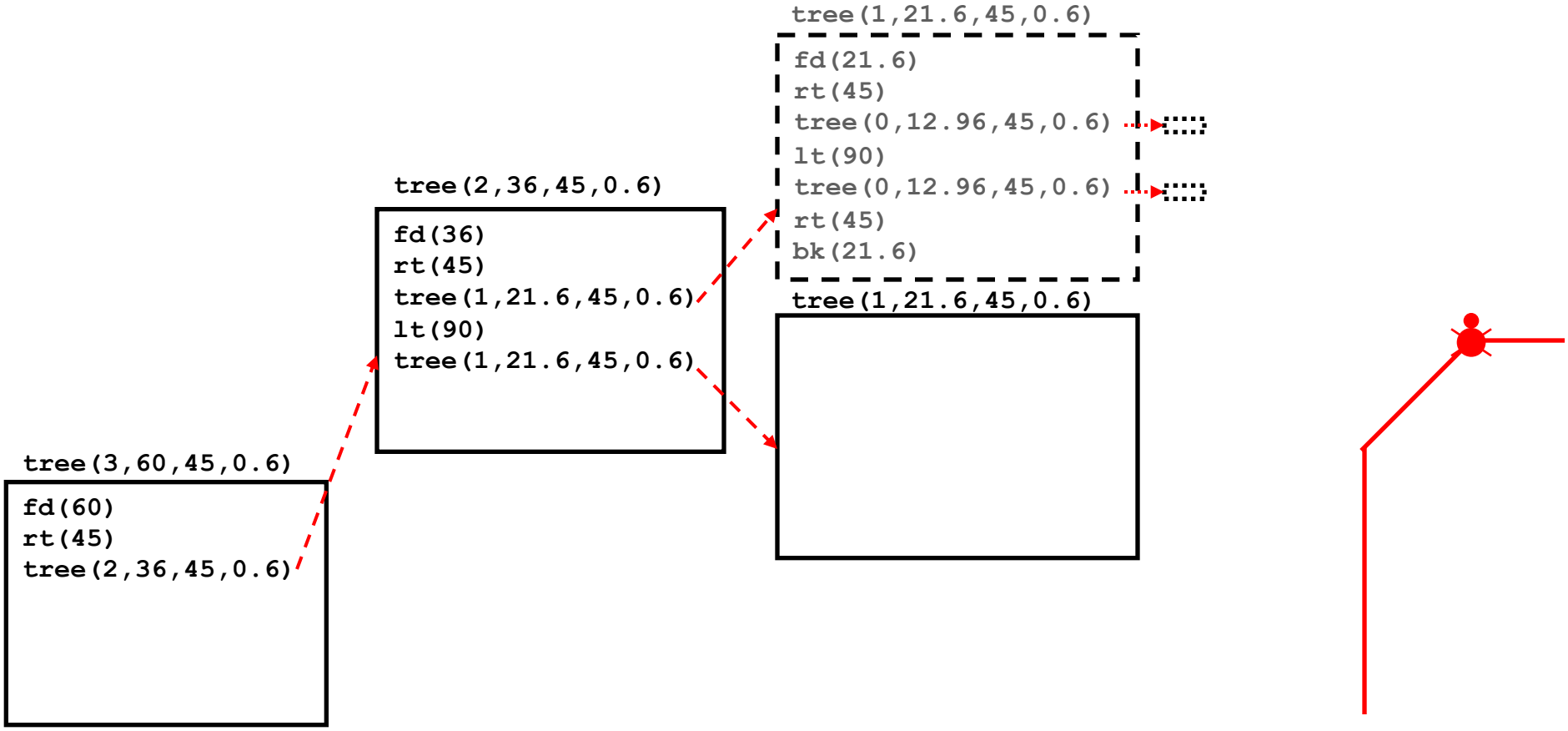
Complete level 0 tree and return to starting position of level 1 tree



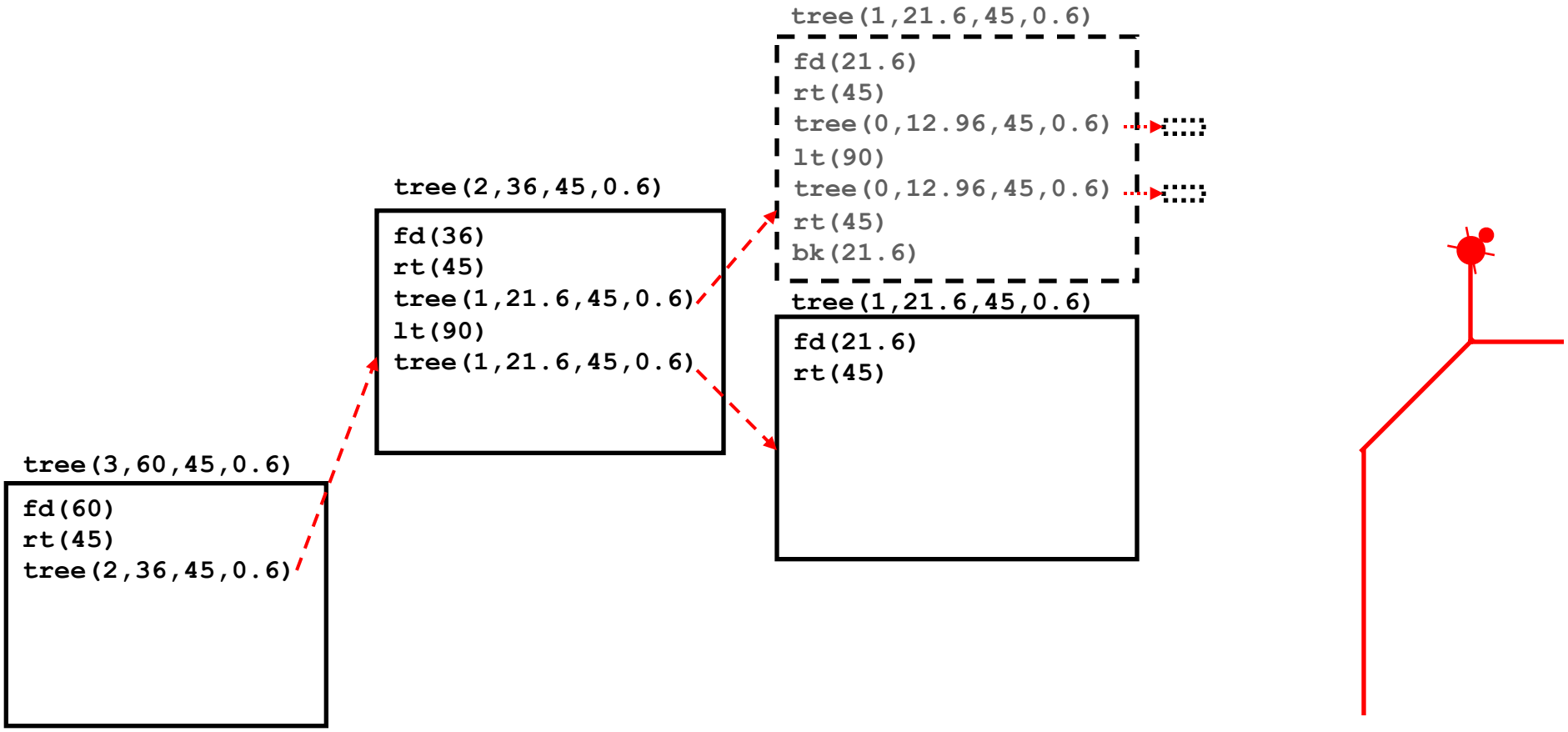
Complete level 1 tree and turn to draw another level 1 tree



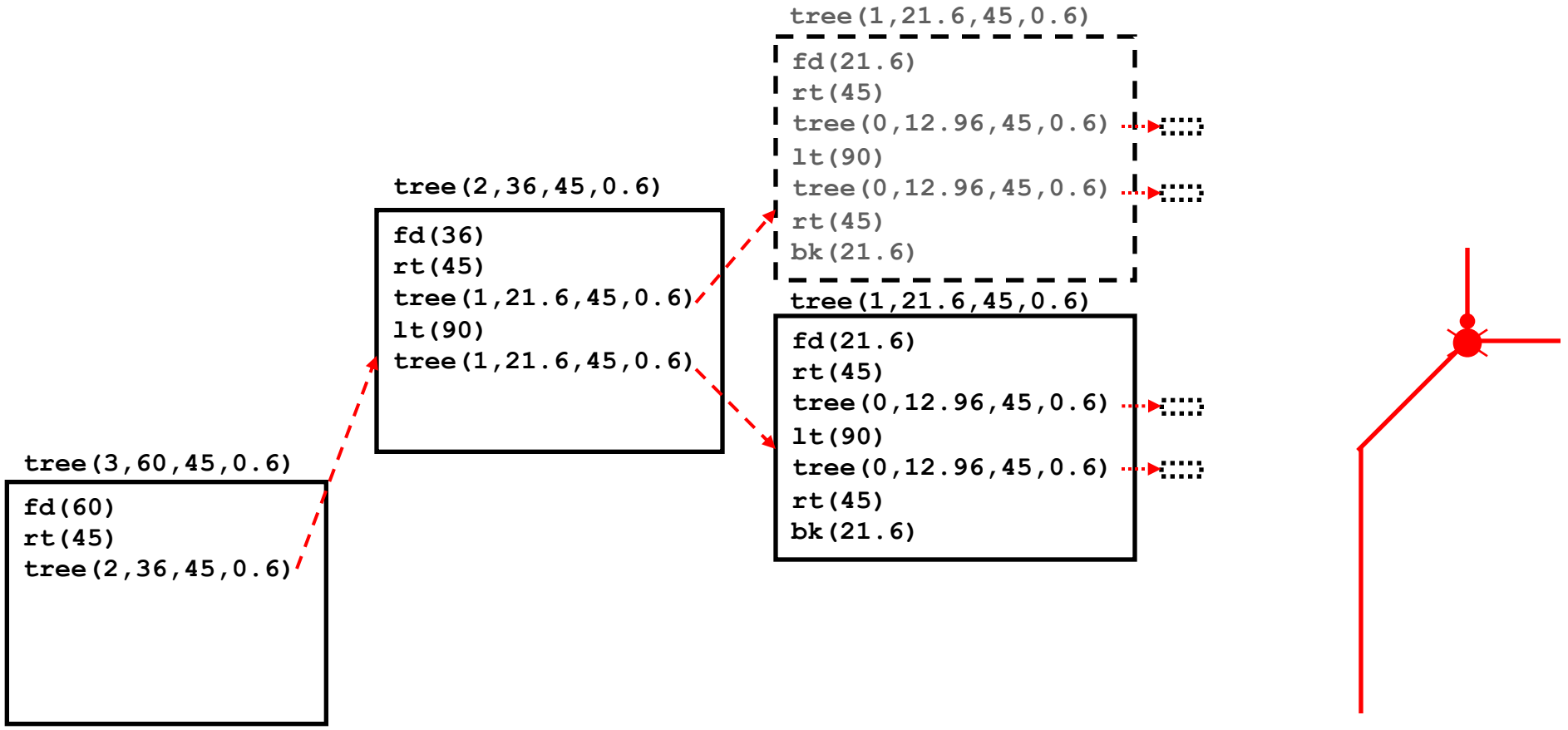
Begin recursive invocation to draw level 1 tree



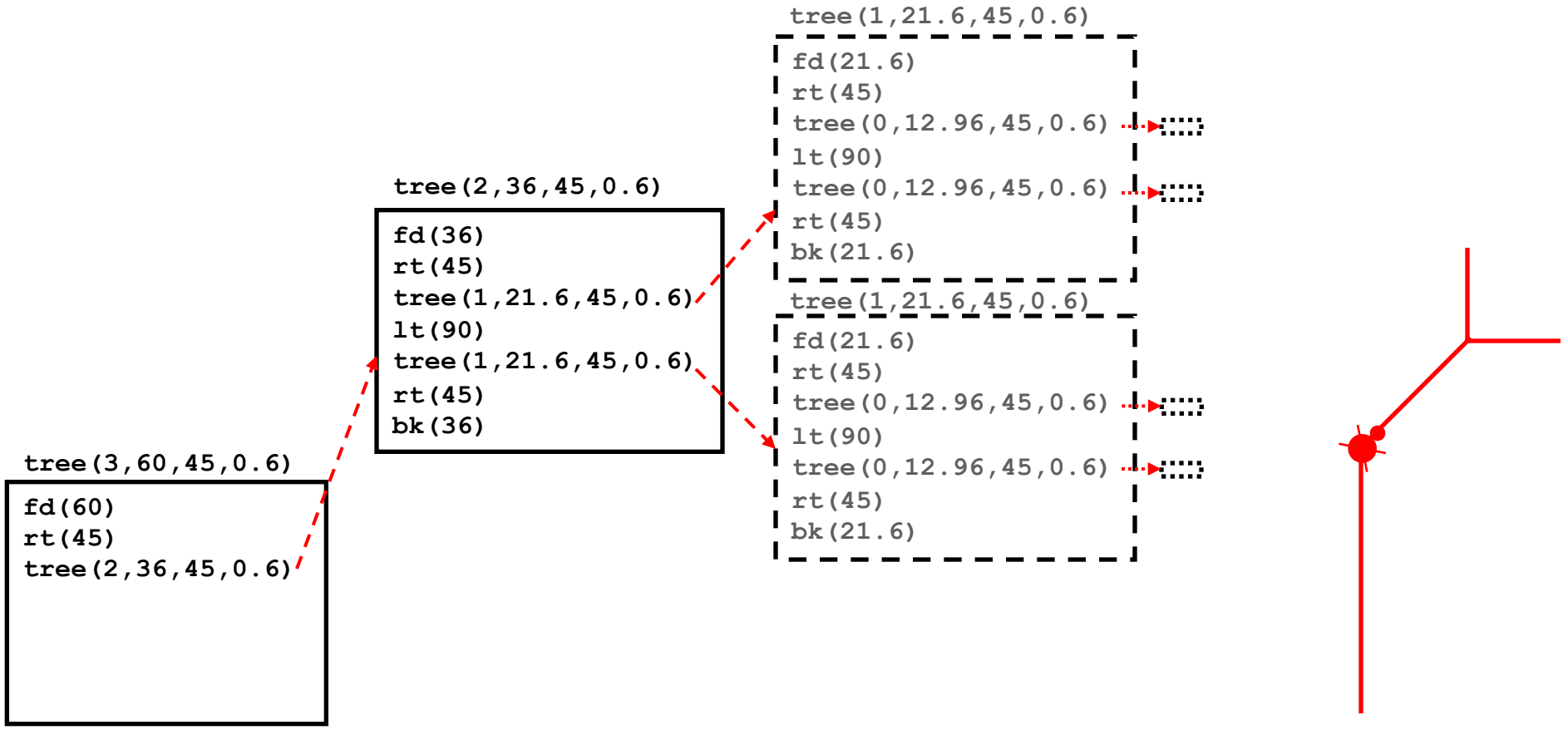
Draw trunk and turn to draw level 0 tree



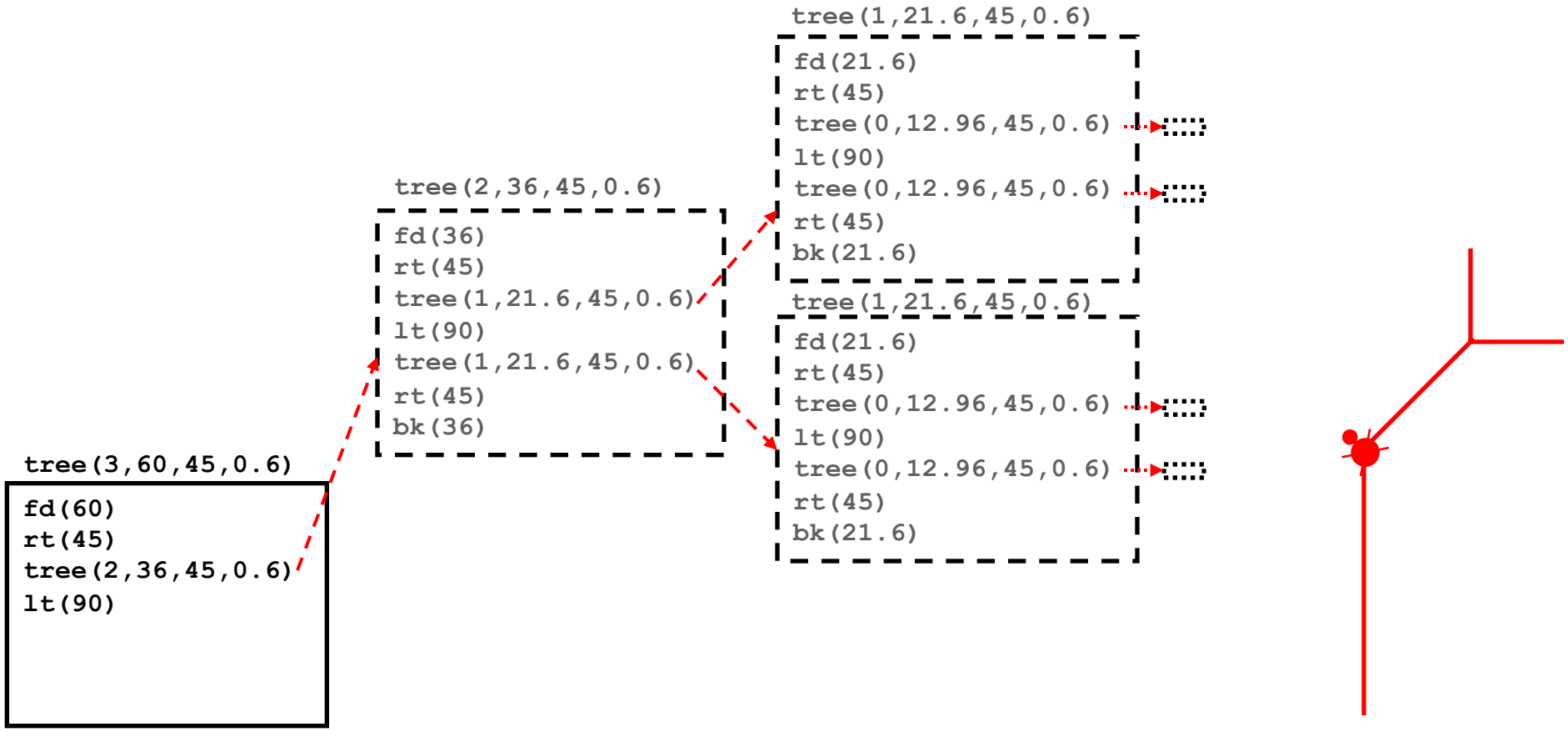
Complete two level 0 trees and return to starting position of level 1 tree



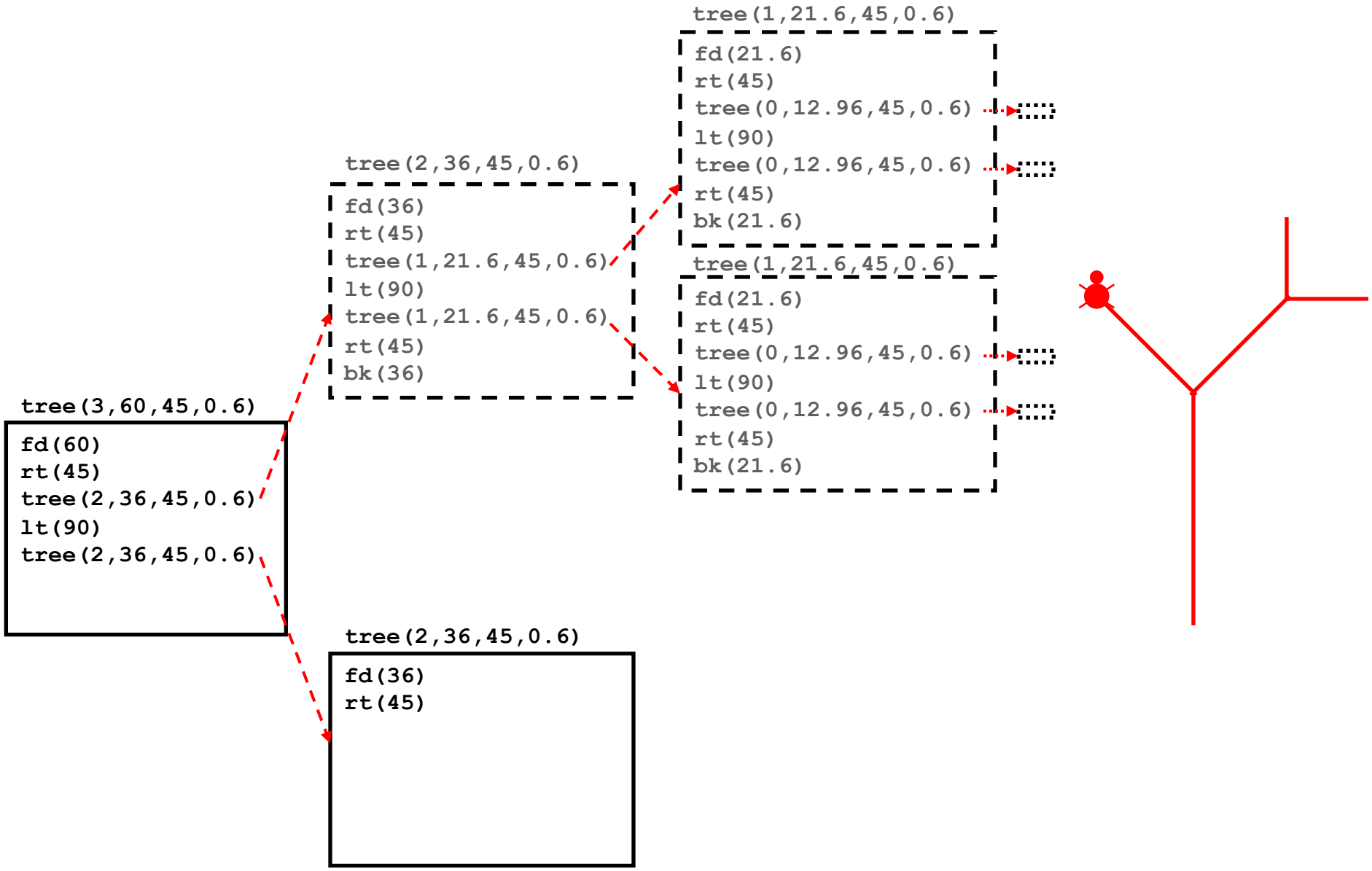
Complete level 1 tree and return to starting position of level 2 tree



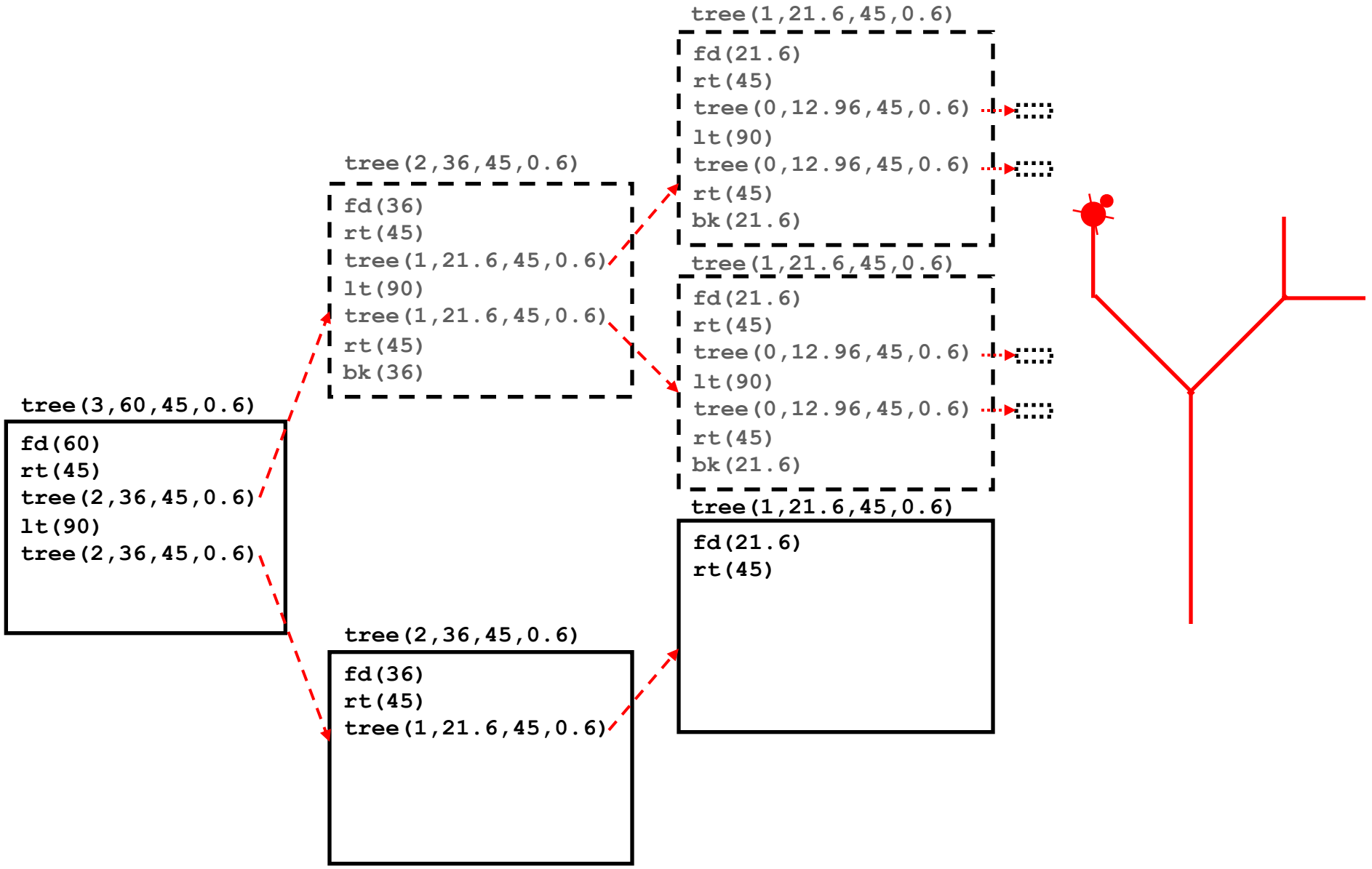
Complete level 2 tree and turn to draw another level 2 tree



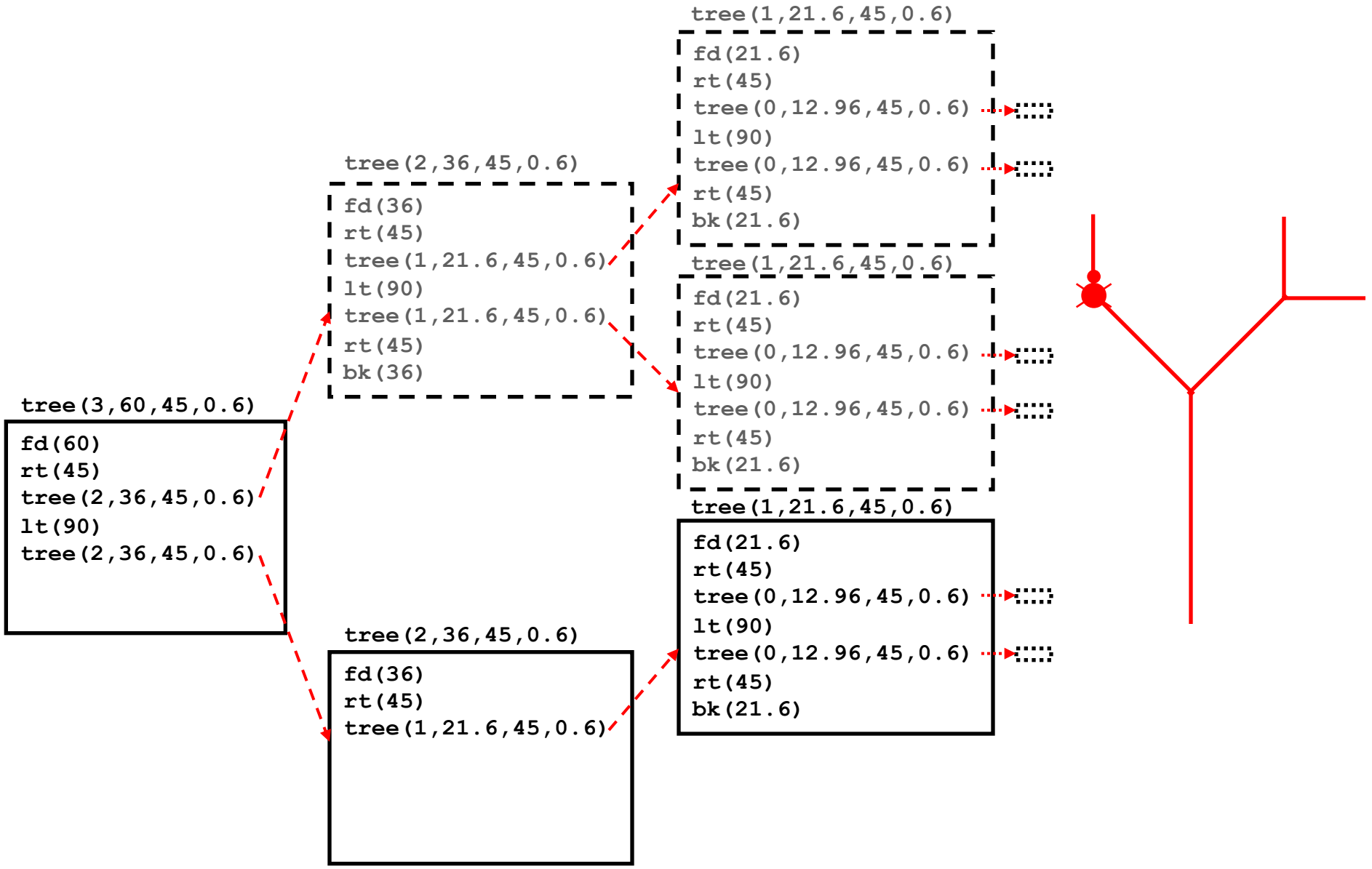
Draw trunk and turn to draw level 1 tree



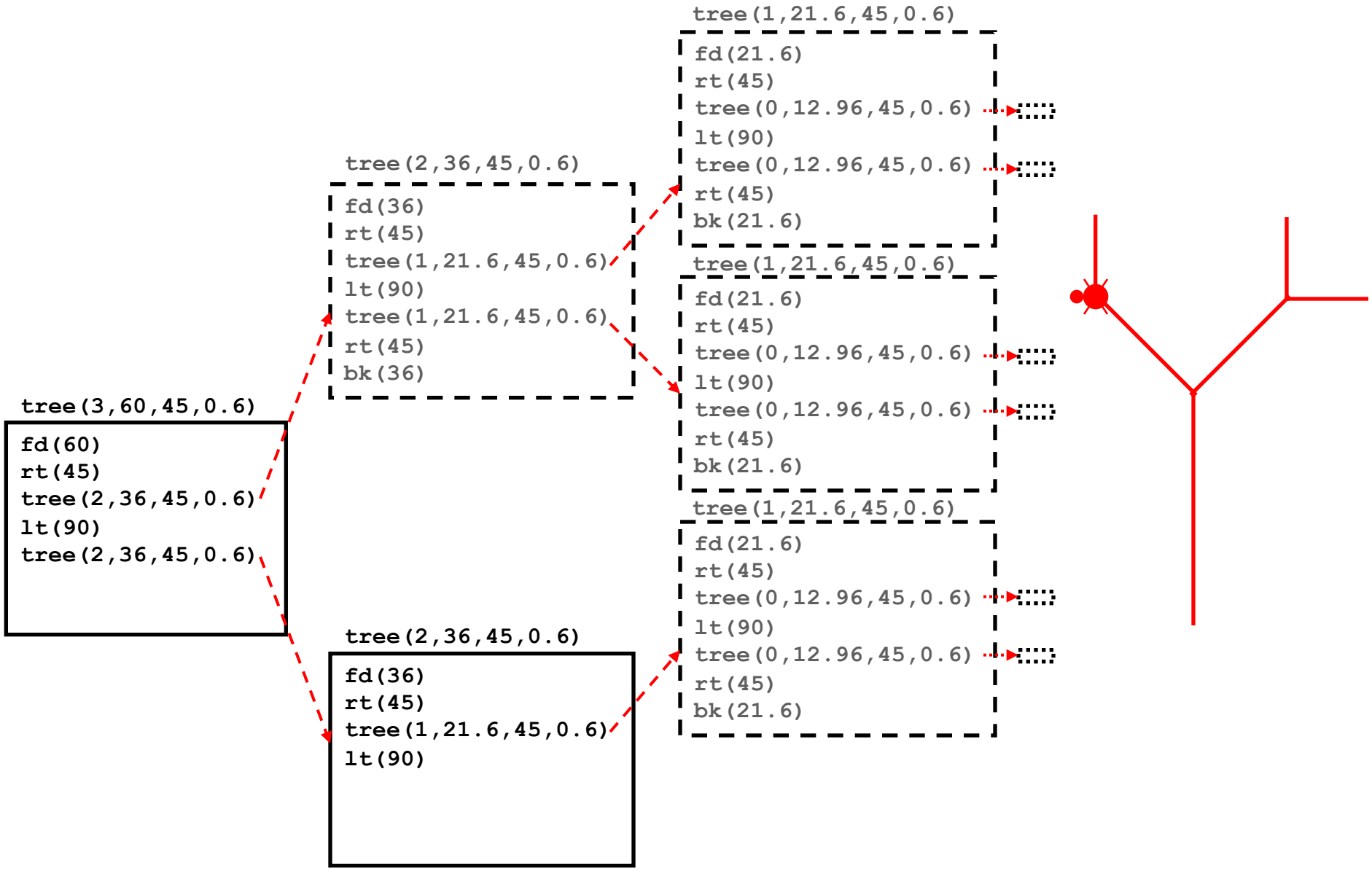
Draw trunk and turn to draw level 0 tree



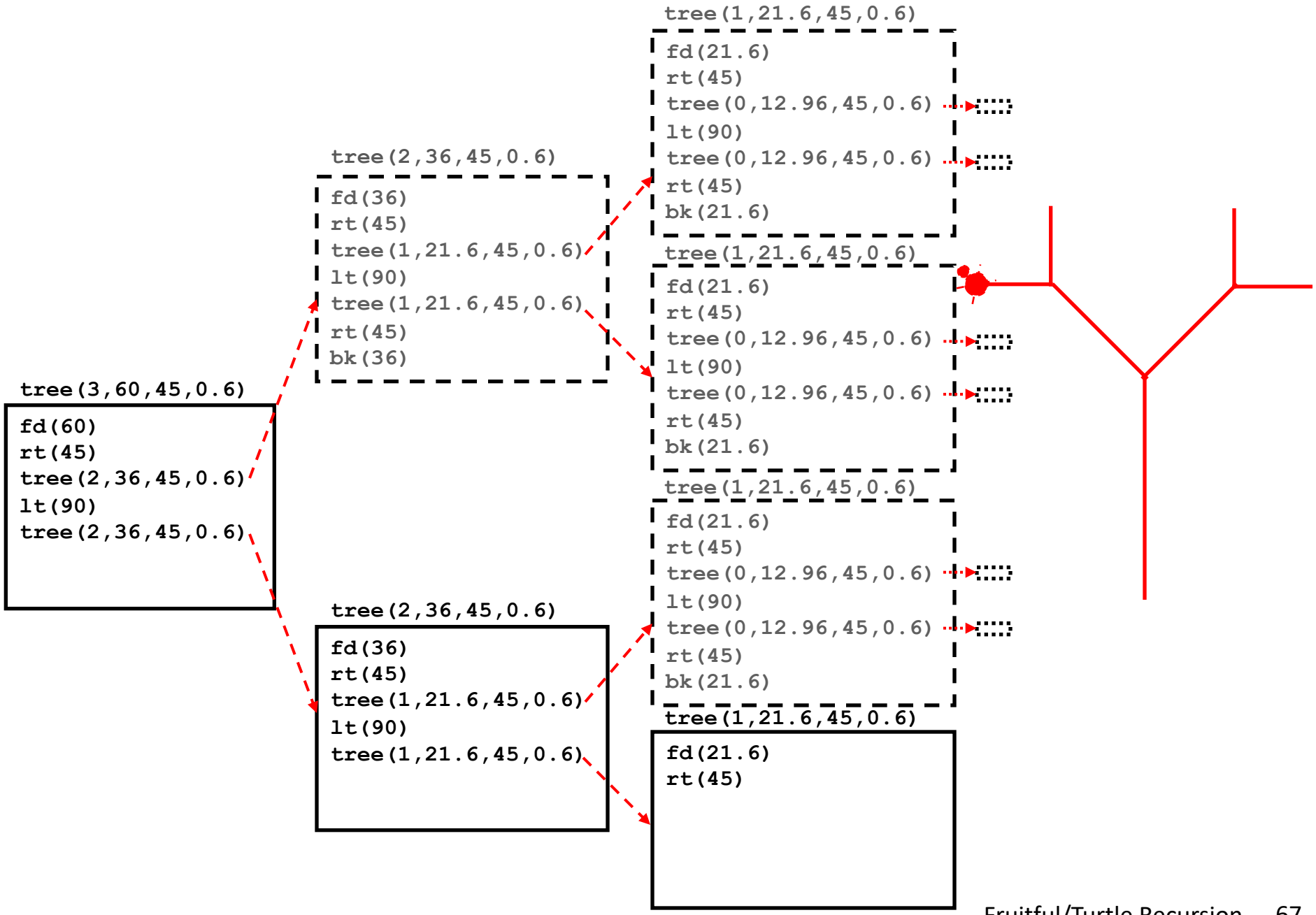
Complete two level 0 trees and return to starting position of level 1 tree



Complete level 1 tree and turn to draw another level 1 tree



Draw trunk and turn to draw level 0 tree



Complete two level 0 trees and return to starting position of level 1 tree

```
tree(3,60,45,0.6)
fd(60)
rt(45)
tree(2,36,45,0.6)
lt(90)
tree(2,36,45,0.6)
```

```
tree(2,36,45,0.6)
fd(36)
rt(45)
tree(1,21.6,45,0.6)
lt(90)
tree(1,21.6,45,0.6)
rt(45)
bk(36)
```

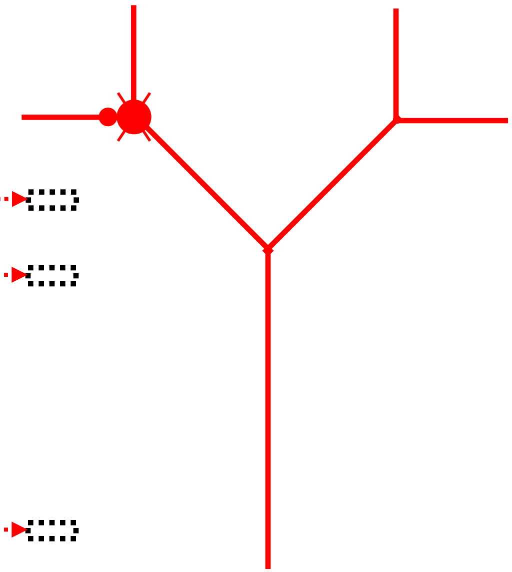
```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

```
tree(2,36,45,0.6)
fd(36)
rt(45)
tree(1,21.6,45,0.6)
lt(90)
tree(1,21.6,45,0.6)
```

```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```



Complete level 1 tree and return to starting position of level 2 tree

tree(3,60,45,0.6)

```
fd(60)
rt(45)
tree(2,36,45,0.6)
lt(90)
tree(2,36,45,0.6)
```

tree(2,36,45,0.6)

```
fd(36)
rt(45)
tree(1,21.6,45,0.6)
lt(90)
tree(1,21.6,45,0.6)
rt(45)
bk(36)
```

tree(2,36,45,0.6)

```
fd(36)
rt(45)
tree(1,21.6,45,0.6)
lt(90)
tree(1,21.6,45,0.6)
rt(45)
bk(36)
```

tree(1,21.6,45,0.6)

```
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

tree(1,21.6,45,0.6)

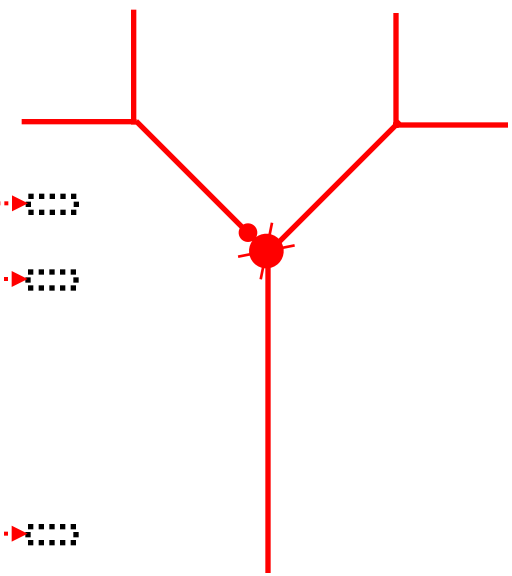
```
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

tree(1,21.6,45,0.6)

```
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

tree(1,21.6,45,0.6)

```
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```



Complete level 2 tree and return to starting position of level 3 tree

```
tree(3,60,45,0.6)
fd(60)
rt(45)
tree(2,36,45,0.6)
lt(90)
tree(2,36,45,0.6)
rt(45)
bk(60)
```

```
tree(2,36,45,0.6)
fd(36)
rt(45)
tree(1,21.6,45,0.6)
lt(90)
tree(1,21.6,45,0.6)
rt(45)
bk(36)
```

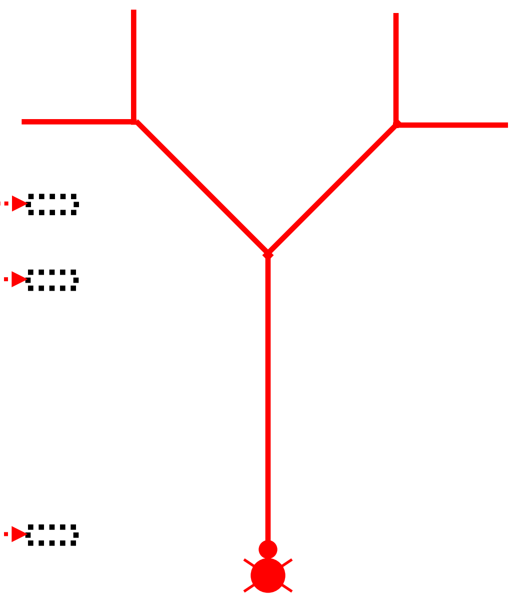
```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

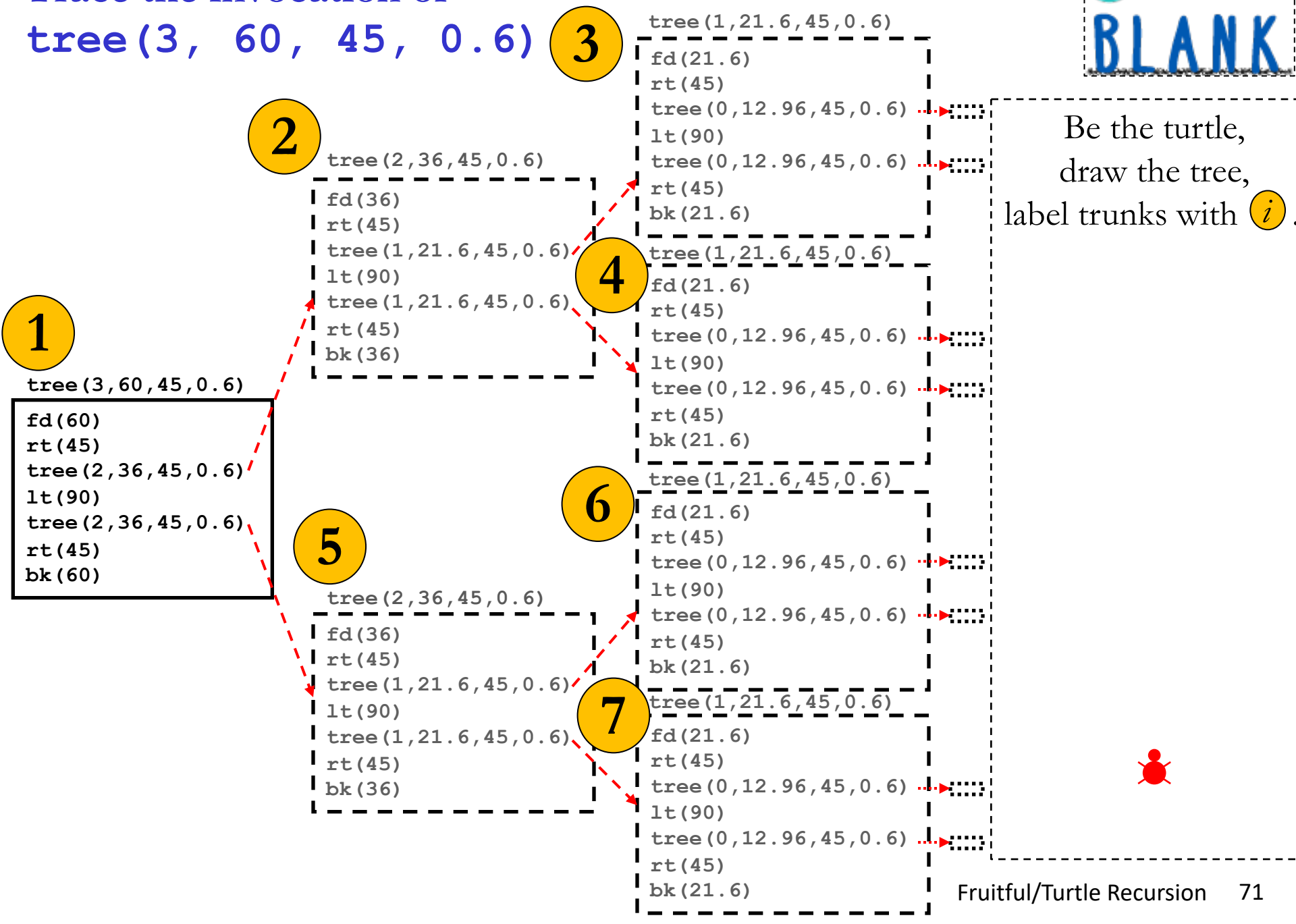
```
tree(2,36,45,0.6)
fd(36)
rt(45)
tree(1,21.6,45,0.6)
lt(90)
tree(1,21.6,45,0.6)
rt(45)
bk(36)
```

```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```

```
tree(1,21.6,45,0.6)
fd(21.6)
rt(45)
tree(0,12.96,45,0.6)
lt(90)
tree(0,12.96,45,0.6)
rt(45)
bk(21.6)
```



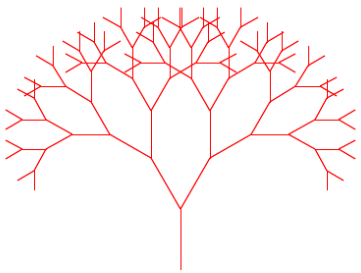
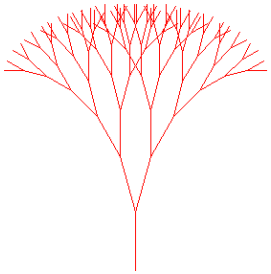
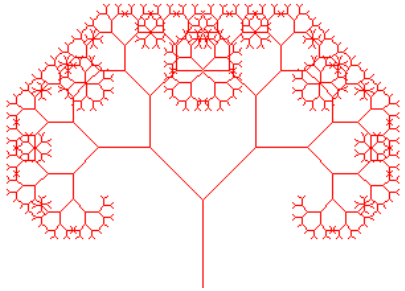
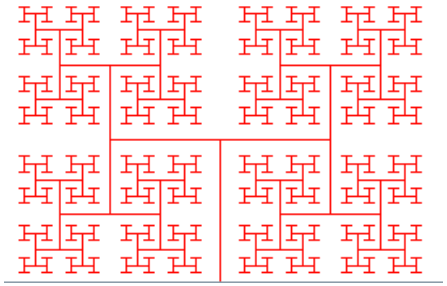
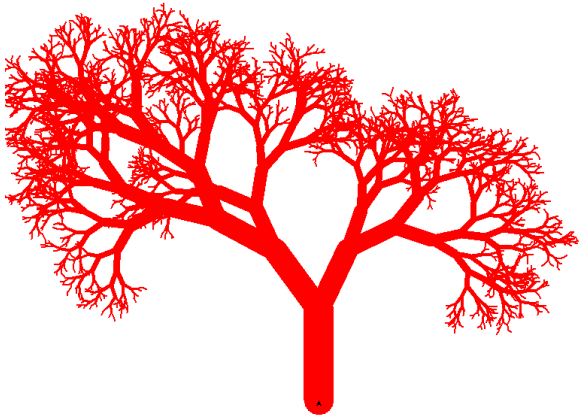
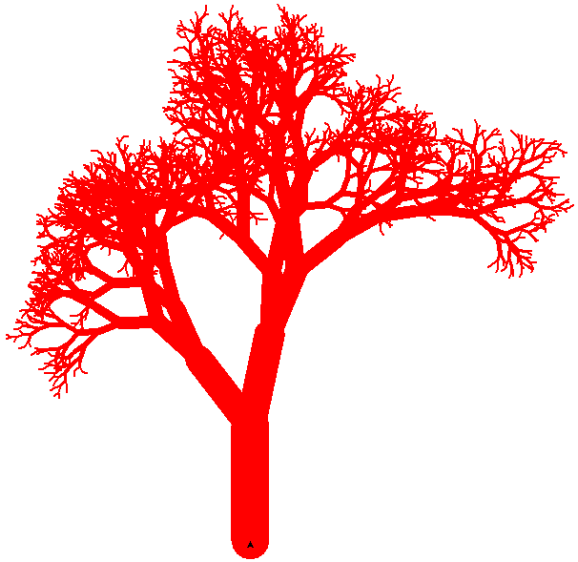
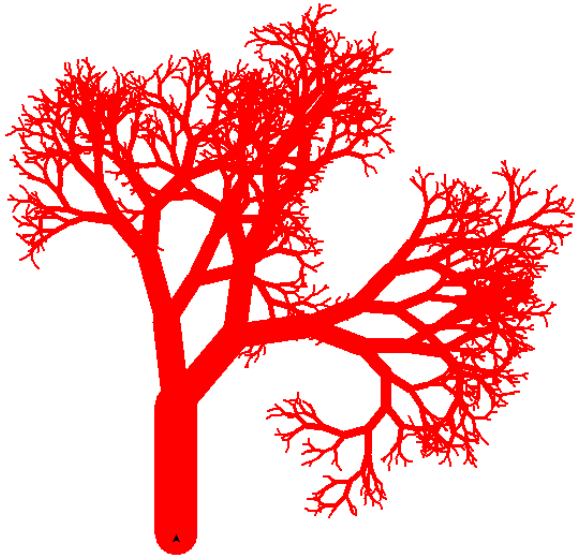
Trace the invocation of `tree(3, 60, 45, 0.6)`



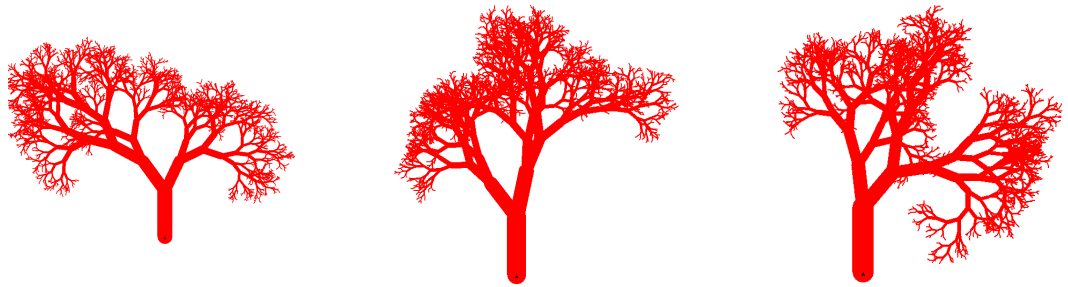
CS 111 Fill in the BLANK

Be the turtle,
draw the tree,
label trunks with *i*.

The squirrels aren't fooled…

Random Trees



```
def treeRandom(length, minLength, thickness, minThickness,
               minAngle, maxAngle, minShrink, maxShrink):
    if (length < minLength) or (thickness < minThickness): # Base case
        pass # Do nothing
    else:
        angle1 = random.uniform(minAngle, maxAngle)
        angle2 = random.uniform(minAngle, maxAngle)
        shrink1 = random.uniform(minShrink, maxShrink)
        shrink2 = random.uniform(minShrink, maxShrink)
        pensize(thickness)
        fd(length)
        rt(angle1)
        treeRandom(length*shrink1, minLength, thickness*shrink1,
                   minThickness, minAngle, maxAngle, minShrink, maxShrink)
        lt(angle1 + angle2)
        treeRandom(length*shrink2, minLength, thickness*shrink2,
                   minThickness, minAngle, maxAngle, minShrink, maxShrink)
        rt(angle2)
        pensize(thickness)
        bk(length)
```