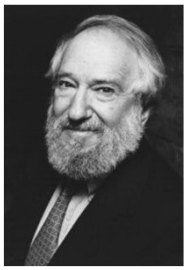
Introducing *Turtle System* and Recursion

Introducing the *Turtle System* and Recursion

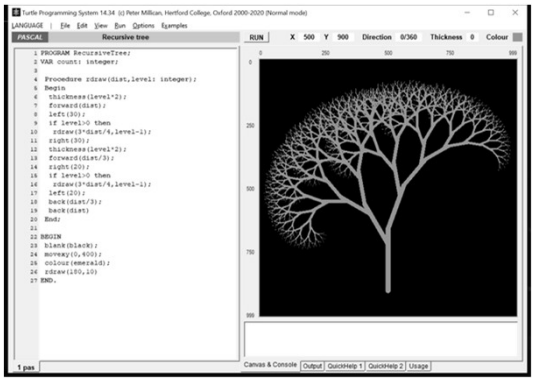
Online UNIQ Summer School, July 2020



Professor Peter Millican
Faculties of Philosophy and Computer Science
Hertford College, Oxford

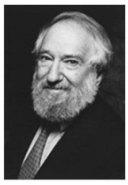
1

1. Introducing the *Turtle System*



2

Papert and Turtle Graphics



- Seymour Papert famously devised the idea of *Turtle Graphics*, whereby students learn to program by giving instructions to an imaginary "turtle" moving around the screen, drawing as it goes.
  - He argued that such learning can enhance general problem-solving skills, not only in programming.
  - The initial focus on *graphics* makes the learning *vivid*, *interactive*, and *relevant* to the student right from the start, with potential for personal *creativity*.
  - All of these features foster effective learning.

3

3

Virtues of Turtle Graphics for Learning Computational Thinking

- Turtle Graphics systems are immensely popular around the world, with well-recognised virtues:
  - Simple, based on an intuitive metaphor;
  - We naturally find visual comprehension far easier and more natural than abstraction;
  - Results of program can literally be *seen*, which helps in debugging and "what if" experimentation (*vastly* easier than with, for example, arithmetical output);
  - Enables complex patterns to be generated;
  - Encourages creativity, and self-pacing;
  - Provides scope for endless challenges.

4

4

The Oxford *Turtle System*

- *Turtle System* is distinctive in several ways:
  - *Starts from* the Turtle metaphor, but moves seamlessly to more advanced concepts *within the same overall structure* (making this easier for students and teachers)
  - Trivially easy to set up and run (and completely free).
  - More than 100 built-in examples, covering a wide range.
  - Specially designed to make interesting types of program easy to implement (e.g. recursive and other patterns, animations, interactive games, cellular automata, scientific and mathematical models).
  - Incorporates multiple programming languages, enabling students to learn about their differences.

5

5

"But Are These *Real*, Industrial-Scale, Programming Languages?"

- *Real* languages: Yes, they are!
- *Industrial-scale*: No, they are not – but that's a Good Thing, not a Bad Thing!!
  - It's a serious (but common) mistake to think of computer languages as like natural languages (e.g. French), where lots of practice *in that specific language* is vital.
  - By comparison, programming languages are *almost trivially simple*, with very few words and clear structures. The crucial obstacle is instead learning to *think computationally*, and once you can do that well, it's easy to transfer this skill to a different programming language.

6

6

Introducing *Turtle System* and Recursion

Virtues of the Barebones Approach

- Starting with a *limited programming language subset* has several major advantages:
  - Easy to learn the basic structures quickly (which is also great for busy teachers, tutors, parents etc.).
  - Learning is focused on mastering key principles rather than learning endless details of a large system.
  - Unlike a full-scale system (e.g. Java or Python), *you are forced to solve low-level computational problems yourself*, rather than finding an "off the shelf" module that someone else has written (which is efficient for an employed programmer, but *hopeless* for learning).
  - Error messages can be precisely targeted, which is impossible in an "industrial-scale" system.

7

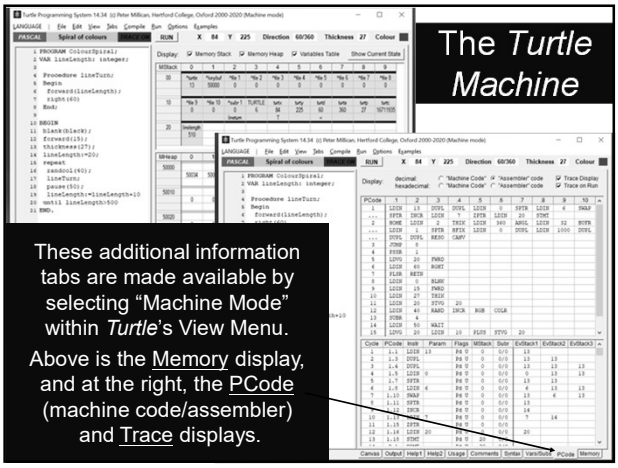
7

Deeper Understanding of Computing

- Because the system is uncomplicated, it can run on a (virtual) *Turtle Machine* designed to be simple enough for clear display and self-learning.
  - The virtual machine is also relatively easy to implement on different devices (e.g. PC, web).
- Source code (written in a "high-level" language such as BASIC, Pascal, or Python) is compiled to machine code for the Turtle Machine:
  - Machine/assembler code can be inspected and understood, traced in real time, etc.
  - Memory can be inspected, with help to see dynamic treatment of variables etc.

8

8

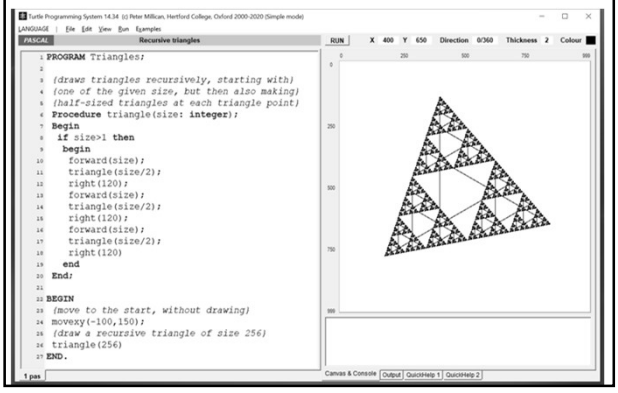


9



10

2. Introducing Recursion



```
PROGRAM Triangles;

(draws triangles recursively, starting with)
(one of the given size, but then also making)
(half-sized triangles at each triangle point)
Procedure triangle(size: integer);
Begin
 if size>1 then
  begin
   forward(size);
   triangle(size/2);
   right(120);
   forward(size);
   triangle(size/2);
   right(120);
   forward(size);
   triangle(size/2);
   right(120)
  end
End;

BEGIN
 (move to the start, without drawing)
 movexy(-100,150);
 (draw a recursive triangle of size 256)
 triangle(256)
END.
```

11

Example: Exploring Triangles

```
PROGRAM Triangle1;
BEGIN
 movexy(-100,150); {move to the start, without drawing}
 forward(256);     {draw the first side, and turn}
 right(120);
 forward(256);     {draw the second side, and turn}
 right(120);
 forward(256);     {draw the third side, and turn}
 right(120)
END.
```

12

12

Introducing *Turtle System* and Recursion

A Triangle Procedure

```
Procedure triangle(size: integer);
 Begin
  if size>1 then
   begin
    forward(size);
    right(120);
    forward(size);
    right(120);
    forward(size);
    right(120)
   end
 End;
```

{size parameter is set by procedure call}

{proceed to draw only if size>1}

"Guarding" the procedure with an "if … then …" condition is crucial if it is to be made recursive. Without such a guard, a recursive version of the procedure would go on calling itself repeatedly, getting into an infinite recursive loop without any termination condition.

13

13

Making the Procedure Recursive

```
Procedure triangle(size: integer);
 Begin
  if size>1 then
   begin
    forward(size);  triangle(size/2);
    right(120);
    forward(size);  triangle(size/2);
    right(120);
    forward(size);  triangle(size/2);
    right(120)
   end
 End;
```

With these inserts, the procedure draws a half-size triangle after reaching each corner in turn. The result is shown in the earlier screen shot.

14

14

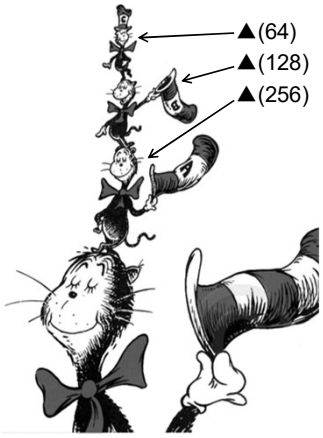
What is Recursion?

- A *recursive procedure* or *function* is one which "calls itself" – in the case we've just seen, the *triangle* procedure (assuming it is given a size value greater than 1) breaks off each time it reaches a new corner, and calls itself to draw a triangle of half the size.
- Thus *triangle*(256) calls *triangle*(128), which in turn calls *triangle*(64), and so on down to *triangle*(4), *triangle*(2), and finally *triangle*(1), which does nothing.
- Each of these procedures pauses while the smaller-valued procedures are in progress. Control jumps back to each "parent" procedure (e.g. *triangle*(16)) when its "child" procedure (e.g. *triangle*(8)) completes.

15

15

- For understanding recursion, it's helpful to think of The Cat in the Hat and his assistants.
- He operates the main program; they do the procedures.
  - Each assistant cat jumps down to do its work when it is called by the more senior cat on whose head it sits.
  - When it has finished, it jumps back up, and the more senior cat (who has meanwhile been waiting) continues.



16

16

Recursive Calculation

In a recursive function, the commonest pattern is that values for higher input values (called *parameters* or *arguments*) are calculated in terms of the values for lower input values. Take, for instance, the factorial function (e.g. factorial(5), or "5!", equals 5 x 4 x 3 x 2 x 1 = 120):

```
Function factorial(n: integer): integer;
Begin
 if n=0 then
  result:=1
 else
  result:=n*factorial(n-1)
End;
```

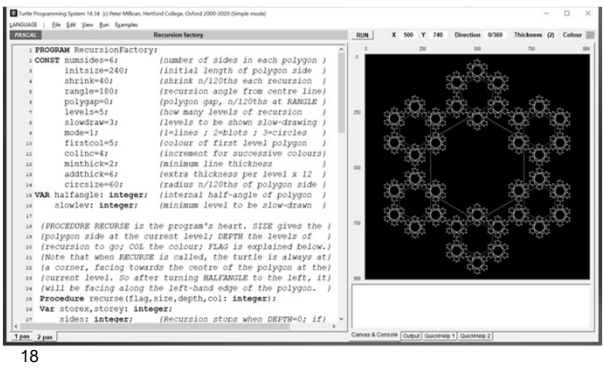
factorial(0) is defined as equal to 1

e.g. factorial(5) equals 5 x factorial(4)

17

17

3. Playing with Recursion



18

18

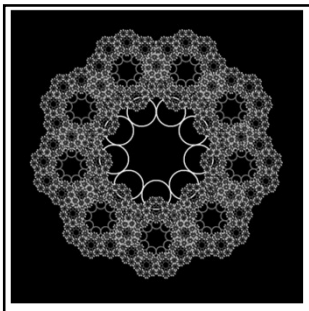
Introducing *Turtle System* and Recursion

The Recursion Factory Program

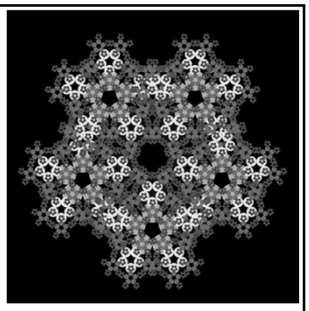
- This program, filed under "self-similarity" in the Examples menus, is designed to enable you to specify a huge variety of recursive shape patterns by setting constants to determine such things as:
  - The number of sides in each shape (e.g. 6 for hexagons)
  - The initial size, and rate of shrinkage at each level
  - The angle between patterns at adjacent levels
  - The gap between polygons at adjacent levels
  - The number of levels of recursion
  - Whether to draw lines, or blots, or circles
  - The sequence of colours to be used at different levels
  - The sequence of line thicknesses at different levels.

19

19



- numsides = 9
- shrink = 40
- polygap = 120
- mode = 3
- firstcol = 4



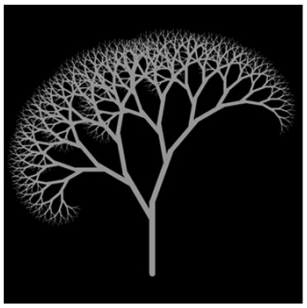
- numsides = 5
- shrink = 60
- polygap = 60
- mode = 2
- firstcol = 3

20

20

Designing Trees

The Recursive Tree program illustrates a different recursive drawing technique. It's an interesting challenge to try to produce your own pretty trees using a similar method.

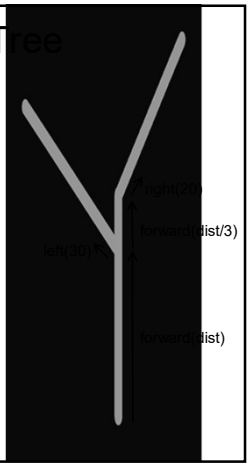


21

21

One Level of the Tree

- At each level, movement and drawing take place as shown here. Starting at the bottom, the Turtle moves forward by *dist* (red arrow), then turns left by 30 degrees (yellow arrow).
- It then turns back by 30 degrees, moves forward by *dist/3* (red arrow), and turns right by 20 degrees (yellow arrow).
- It then retraces its steps to get back to the starting point.
- The two yellow arrows indicate points where a recursive call can take place to the next level.



22

22

```
Procedure rdraw(dist,level: integer);
Begin
 thickness(level*2);
 forward(dist);
 left(30);
 if level>0 then
  rdraw(3*dist/4,level-1);
 right(30);
 thickness(level*2);
 forward(dist/3);
 right(20);
 if level>0 then
  rdraw(3*dist/4,level-1);
 left(20);
 back(dist/3);
 back(dist)
End;
```


if level>9


if level>8


if level>7


if level>6

23

23

Self-Similarity by Other Means

Recursive Drawing



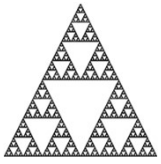
Recursive Deletion



Iterated Dots



Iterated Copying



24

24