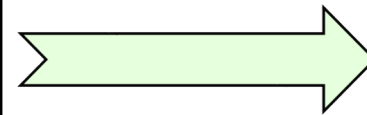


# Drawing Heighway's Dragon Recursive Function Rewrite From Imperative Style in Pascal 64 To Functional Style in Scala 3

  
Pascal



 Scala



slides by



@philip\_schwarz



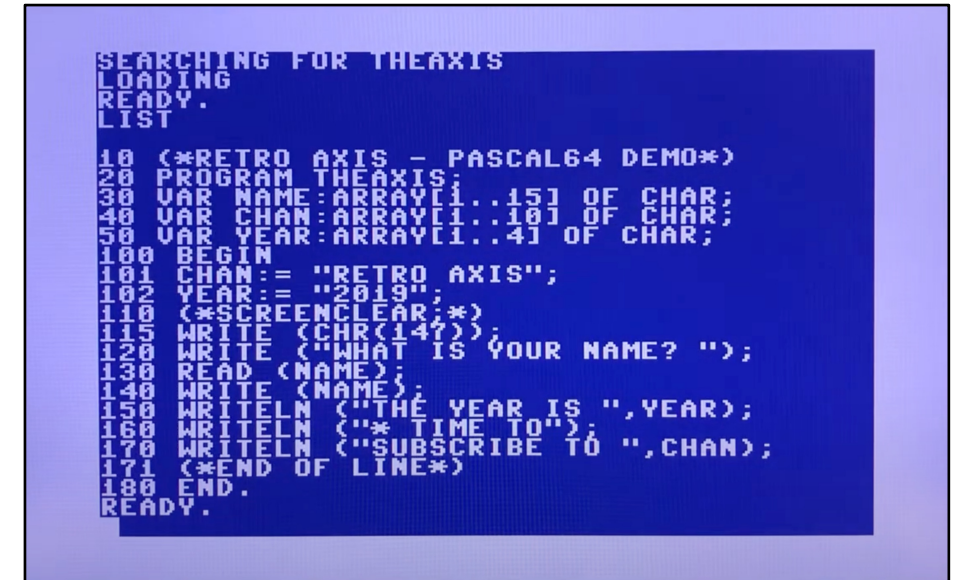
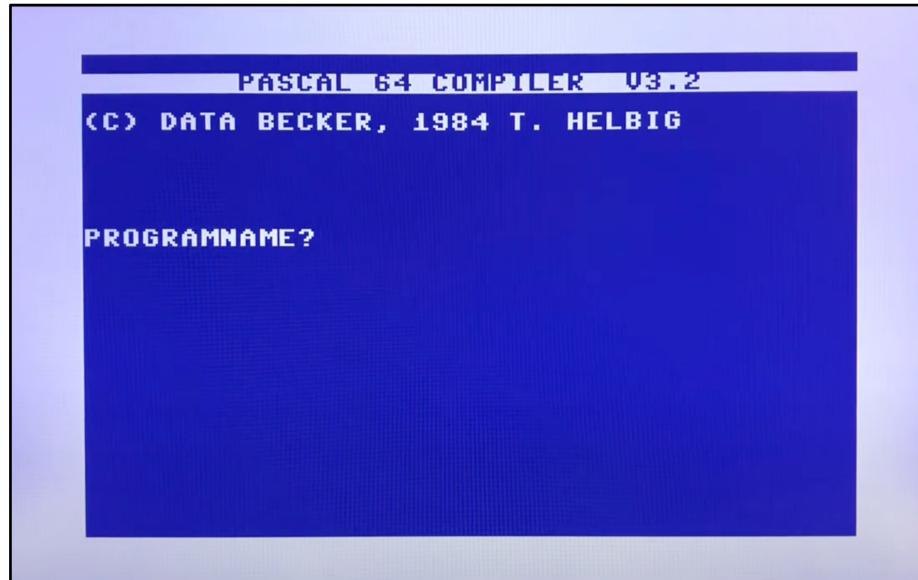
<https://fpilluminated.org/>



The first two **programming languages** that I studied as part of my computer science degree (back in 1985), were **Algol 68** and **Pascal**.

It was only at that time that I realised that a **Pascal** compiler existed for the **Commodore 64**, the computer which I used as a teenager.

See below for an example of what the simplest **Pascal** program might have looked like using the **Pascal 64** compiler.



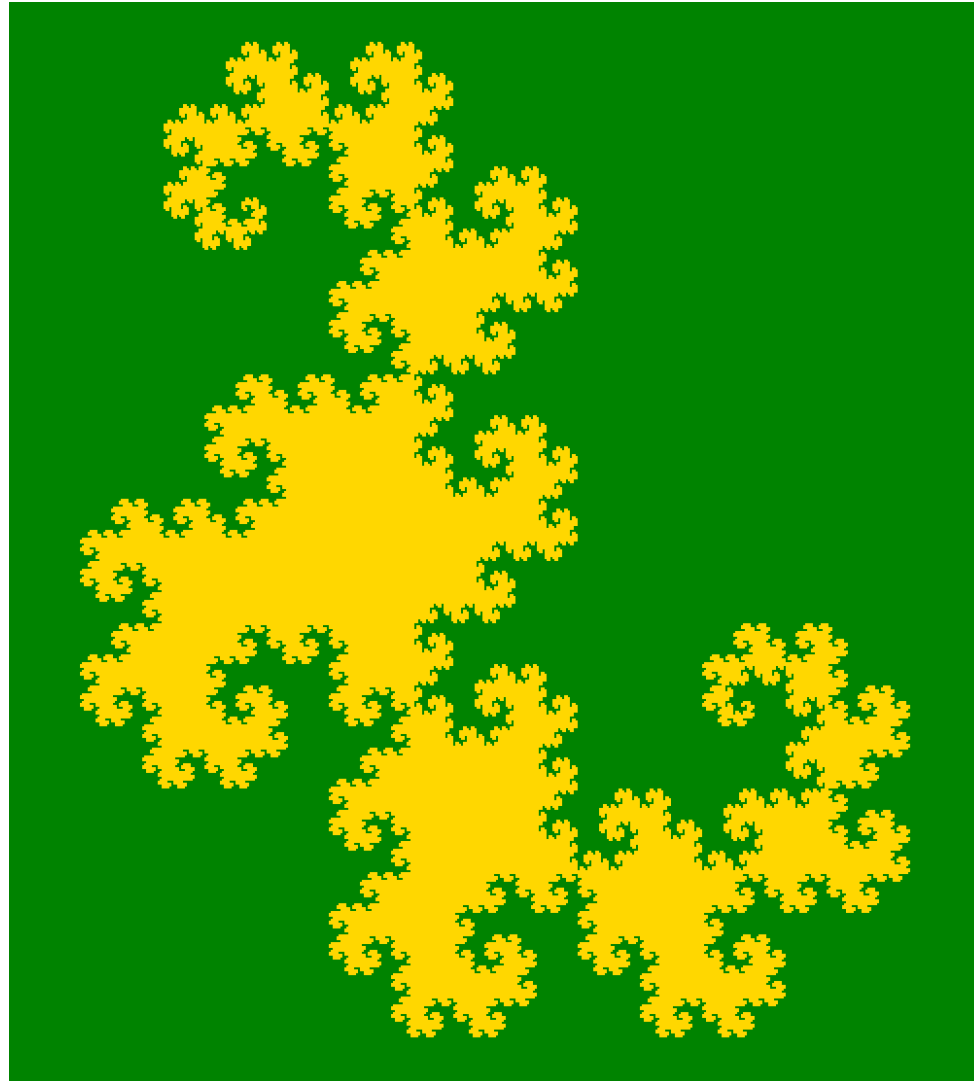
screenshots from youtube video by <https://www.retroaxis.tv/>



Now and again I bump into a four-decades old printout of one of the first **Pascal** programs that I ran when I bought the **Pascal 64** compiler.

The program draws a **Heighway Dragon** on the screen.

Here is what a **dragon aged 17** looks like when drawn using **lines** of the shortest possible **length**.





And here are screenshots of a faded printout of the program's code. On the next slide I type out the code again for clarity.

```
- PROGRAM DRAWING HEIGHWAY'S DRAGON -  
  
READY.  
  
100 PROGRAM FRACTALS;  
130 VAR X1,X2,Y1,Y2,DAY,LENGTH:INTEGER;  
150 PROCEDURE DRAW (X1,Y1:REAL; X2,Y2:INTEGER);  
160 VAR COUNTER,DX,DY:INTEGER;  
170 VAR STEP:REAL;  
180 BEGIN  
190 ; X1:= TRUNC(X1);  
200 ; Y1:= TRUNC(Y1);  
210 ; DX:= X2-X1;  
220 ; DY:= Y2-Y1;  
230 ; IF (X1=X2) AND (Y1=Y2)  
240 ; THEN PLOT X1,Y1;  
250 ; ELSE  
260 ; BEGIN  
270 ; IF ABS(DX)>ABS(DY)  
280 ; THEN  
290 ; BEGIN  
300 ; STEP:= DY/ABS(DX);  
310 ; COUNTER:= DX/ABS(DX);  
320 ; WHILE X1<>X2  
330 ; DO  
340 ; BEGIN  
350 ; PLOT X1,Y1;  
360 ; Y1:= Y1+STEP;  
370 ; X1:= X1+COUNTER;  
380 ; END;  
390 ; END;  
400 ; ELSE  
410 ; BEGIN  
420 ; STEP:= DX/ABS(DY);  
430 ; COUNTER:= DY/ABS(DY);  
440 ; WHILE Y1<>Y2  
450 ; DO  
460 ; BEGIN  
470 ; PLOT X1,Y1;  
480 ; X1:= X1+STEP;  
490 ; Y1:= Y1+COUNTER;  
500 ; END;  
510 ; END;  
520 ; END;  
530 END;
```

```
540 PROCEDURE DRAGON (DAY:INTEGER; CELL:CHAR);  
550 BEGIN  
560 ; IF DAY=0  
570 ; THEN  
580 ; BEGIN  
590 ; CASE CELL OF  
600 ; "N": Y2:= Y1-LENGTH;  
610 ; "S": Y2:= Y1+LENGTH;  
620 ; "E": X2:= X1+LENGTH;  
630 ; "W": X2:= X1-LENGTH;  
640 ; END;  
650 ; DRAW (X1,Y1,X2,Y2);  
660 ; X1:= X2;  
670 ; Y1:= Y2;  
680 ; END;  
690 ; ELSE  
700 ; BEGIN  
710 ; CASE CELL OF  
720 ; "N": BEGIN  
730 ; DRAGON (DAY-1,"W");  
740 ; DRAGON (DAY-1,"N");  
750 ; END;  
760 ; "S": BEGIN  
770 ; DRAGON (DAY-1,"E");  
780 ; DRAGON (DAY-1,"S");  
790 ; END;  
800 ; "E": BEGIN  
810 ; DRAGON (DAY-1,"E");  
820 ; DRAGON (DAY-1,"N");  
830 ; END;  
840 ; "W": BEGIN  
850 ; DRAGON (DAY-1,"W");  
860 ; DRAGON (DAY-1,"S");  
870 ; END;  
880 ; END;  
885 ; END;  
890 END;  
900 BEGIN  
905 ; WRITELN ("HEIGHWAY'S DRAGON.");  
906 ; WRITELN;  
907 ; WRITELN ("INPUT AGE (0-15).");  
908 ; READLN (DAY);  
909 ; WRITELN;  
910 ; WRITELN ("INPUT LENGTH (2-10).");  
911 ; READLN (LENGTH);  
912 ; WRITELN;  
913 ; WRITELN ("INPUT COORDINATES.");  
914 ; READLN (X1,Y1);  
915 ; X2:= X1; Y2:= Y1;  
930 ; GRAPHIC 1; SCREENCLEAR;  
940 ; DRAGON (DAY,"E");  
950 ; REPEAT UNTIL PEEK(197)<>64;  
960 END.
```

```

100 PROGRAM FRACTALS;
130 VAR X1, X2, Y1, Y2, DAY, LENGTH: INTEGER;
150 PROCEDURE DRAW(X1, Y1: REAL; X2, Y2: INTEGER);
160 VAR COUNTER, DX, DY: INTEGER
170 VAR STEP: REAL
180 BEGIN
190 ; X1:= TRUNC(X1);
200 ; Y1:= TRUNC(Y1);
210 ; DX:= X2 - X1;
220 ; DY:= Y2 - Y1;
230 ; IF (X1 = X2) AND (Y1 = Y2)
240 ; THEN PLOT X1,Y1;
250 ; ELSE
260 ; BEGIN
270 ; IF ABS(DX) > ABS(DY)
280 ; THEN
290 ; BEGIN
300 ; STEP:= DY / ABS(DX);
310 ; COUNTER:= DX / ABS(DX);
320 ; WHILE X1 <> X2
330 ; DO
340 ; BEGIN
350 ; PLOT X1,Y1;
360 ; Y1:= Y1 + STEP;
370 ; X1:= X1 + COUNTER;
380 ; END;
390 ; END;
400 ; ELSE
410 ; BEGIN
420 ; STEP:= DX / ABS(DY);
430 ; COUNTER:= DY / ABS(DY);
440 ; WHILE Y1 <> Y2
450 ; DO
460 ; BEGIN
470 ; PLOT X1,Y1;
480 ; X1:= X1 + STEP;
490 ; Y1:= Y1 + COUNTER;
500 ; END;
510 ; END;
520 ; END;
530 END;

```

```

540 PROCEDURE DRAGON (DAY: INTEGER; CELL: CHAR);
550 BEGIN
560 ; IF DAY=0
570 ; THEN
580 ; BEGIN
590 ; CASE CELL OF
600 ; "N": Y2 := Y1 - LENGTH;
610 ; "S": Y2 := Y1 + LENGTH;
620 ; "E": X2 := X1 + LENGTH;
630 ; "W": X2 := X1 - LENGTH;
640 ; END;
650 ; DRAW(X1, Y1, X2, Y2);
660 ; X1:= X2;
670 ; Y1:= Y2;
680 ; END;
690 ; ELSE
700 ; BEGIN;
710 ; CASE CELL OF
720 ; "N": BEGIN
730 ; DRAGON(DAY-1, "W")
740 ; DRAGON(DAY-1, "N")
750 ; END;
760 ; "S": BEGIN
770 ; DRAGON(DAY-1, "E")
780 ; DRAGON(DAY-1, "S")
790 ; END;
800 ; "E": BEGIN
810 ; DRAGON(DAY-1, "E")
820 ; DRAGON(DAY-1, "N")
830 ; END;
840 ; "W": BEGIN
850 ; DRAGON(DAY-1, "W")
860 ; DRAGON(DAY-1, "S")
870 ; END;
880 ; END;
885 ; END;
890 ; END;

```

```

900 BEGIN
905 ; WRITELN ("HEIGHWAY'S DRAGON.");
906 ; WRITELN;
907 ; WRITELN ("INPUT AGE (0-15).");
908 ; READLN (DAY);
910 ; WRITELN ("INPUT LENGTH (2-10).");
911 ; READLN (LENGTH);
912 ; WRITELN;
913 ; WRITELN ("INPUT COORDINATES.");
914 ; READLN (X1, Y1);
915 ; X2:= X1; Y2:= Y1;
930 ; GRAPHIC 1; SCREENCLEAR;
940 ; DRAGON (DAY,"E");
950 ; REPEAT UNTIL PEEK(197)<>64;
960 END.

```

The code is organised in three sections.

The **first section** defines a **procedure** called **DRAW** which is used to draw a **line** going from one point on the screen to another.

The **last section** queries the user for the **parameters** needed to draw a **dragon**.

The **middle section** is a **procedure** called **DRAGON** that uses the **DRAW** function to draw a **dragon** using the **parameters** specified by the user.





What we are interested in is the **DRAGON procedure**.

For what it achieves, the **procedure** looks fairly simple, which is not surprising since it uses **recursion**.

Still, after simply scanning the code a few times, it is not obvious to me what the **key idea** is that the **procedure** exploits to draw the **dragon**.

Also, the **procedure** is not a **pure function**, not only because it uses the **side-effecting DRAW function** (to draw lines), but also because instead of relying solely on its parameters, the **procedure** also uses several **global variables**.

Here are the **objectives** of this deck series:

- Understand how **DRAGON** works
- Rewrite the **Pascal DRAGON** procedure as a **Scala** function whose logic is organised as an **imperative shell** and a **functional core**
- Use the **Scala function** to draw some **dragons**
- See if we can write an alternative **Scala function** whose workings are simpler to understand

```
540 PROCEDURE DRAGON (DAY: INTEGER; CELL: CHAR);
550 BEGIN
560 ; IF DAY=0
570 ; THEN
580 ; BEGIN
590 ; CASE CELL OF
600 ;     "N": Y2 := Y1 - LENGTH;
610 ;     "S": Y2 := Y1 + LENGTH;
620 ;     "E": X2 := X1 + LENGTH;
630 ;     "W": X2 := X1 - LENGTH;
640 ; END;
650 ; DRAW(X1, Y1, X2, Y2);
660 ; X1:= X2;
670 ; Y1:= Y2;
680 ; END;
690 ; ELSE
700 ; BEGIN;
710 ; CASE CELL OF
720 ;     "N": BEGIN
730 ;         DRAGON(DAY-1, "W")
740 ;         DRAGON(DAY-1, "N")
750 ;     END;
760 ;     "S": BEGIN
770 ;         DRAGON(DAY-1, "E")
780 ;         DRAGON(DAY-1, "S")
790 ;     END;
800 ;     "E": BEGIN
810 ;         DRAGON(DAY-1, "E")
820 ;         DRAGON(DAY-1, "N")
830 ;     END;
840 ;     "W": BEGIN
850 ;         DRAGON(DAY-1, "W")
860 ;         DRAGON(DAY-1, "S")
870 ;     END;
880 ; END;
885 ; END;
890 ; END;
```



Let's take a first look at the **DRAGON** procedure, which is **recursively defined**.

In what follows, by a **line**, we don't mean a **mathematical line** of **infinite length**, but rather a **segment** of such a **line**, i.e. a section of the **line** going from one of its **points** to another.

A **dragon** is drawn by drawing the **lines** connecting a **sequence** of **points** on the **screen**. Let's refer to the **sequence** of **points** as the **dragon's path**.

Global variable **LENGTH**, specified by the user, defines the **length** of **lines**.

**Lines** are drawn using the **DRAW** procedure. They are drawn either **vertically** or **horizontally**, i.e. they **start** at one **point** and **end** at another **point** computed by moving a **distance LENGTH** in one of four **directions**: **North**, **South**, **East** and **West**.

The **DRAGON** procedure maintains **global variable** pair **(X1,Y1)**, which represents the **starting point** of the next **line** to be drawn on the **screen**.

The **direction** used to compute the **end point** of the very first **line** is **East**.

The first parameter of the **DRAGON** procedure is **DAY**, the **age** of the **dragon** expressed as a non-negative number of **days**.

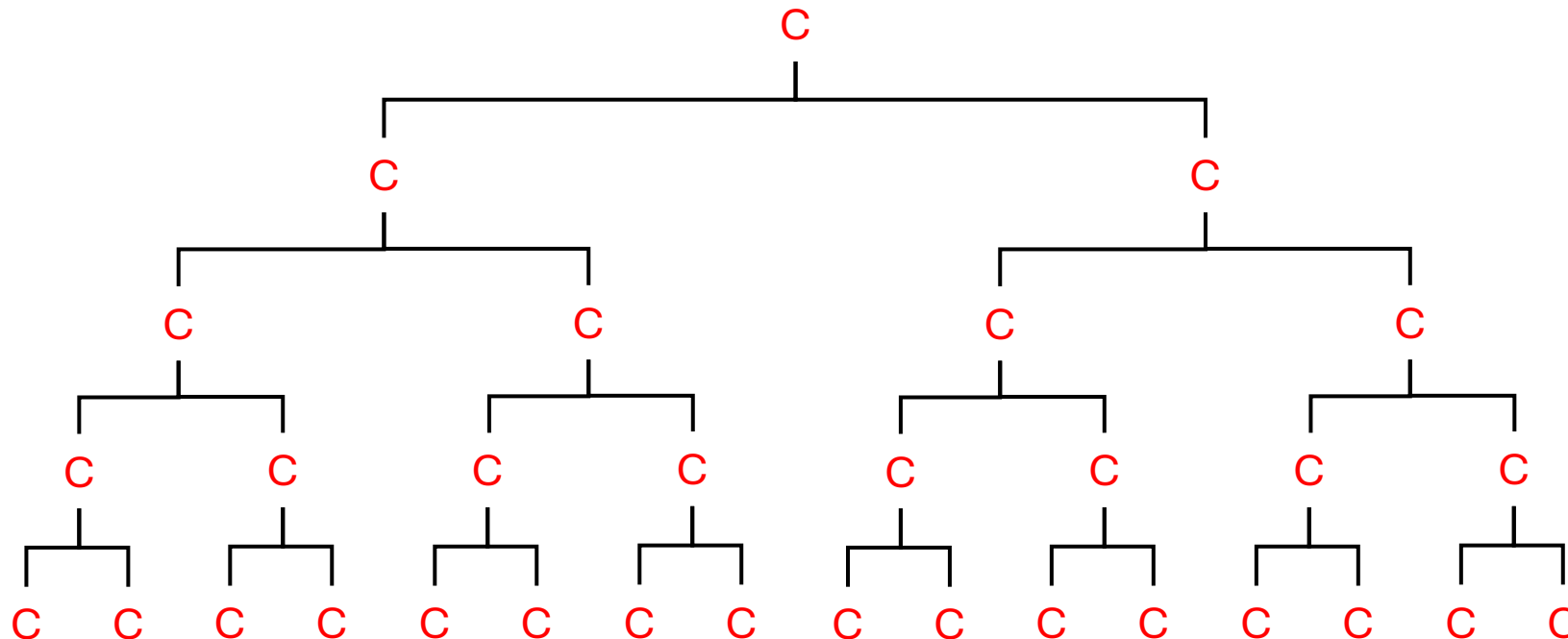
The other parameter is **CELL**, the **direction** of the next **line** to be drawn. Think of the **lines** of a **dragon** as its **cells**, which are drawn by connecting a **starting point (X1,Y1)** to an **end point (X2,Y2)** reached by moving a **distance LENGTH** from the **starting point** in the **direction** indicated by **CELL**.

If an invocation of the **DRAGON** procedure has reached the **base case** of **age zero** then it first **draws** a **line** from **(X1,Y1)** to **(X2,Y2)**, and then sets **(X1,X2)** to **(X2,Y2)**, otherwise the procedure **recursively invokes itself twice** with an age of **DAY - 1**, and with **directions** computed from the given one.

```
540 PROCEDURE DRAGON (DAY: INTEGER; CELL: CHAR);
550 BEGIN
560 ; IF DAY=0
570 ; THEN
580 ; BEGIN
590 ; CASE CELL OF
600 ; "N": Y2 := Y1 - LENGTH;
610 ; "S": Y2 := Y1 + LENGTH;
620 ; "E": X2 := X1 + LENGTH;
630 ; "W": X2 := X1 - LENGTH;
640 ; END;
650 ; DRAW(X1, Y1, X2, Y2);
660 ; X1:= X2;
670 ; Y1:= Y2;
680 ; END;
690 ; ELSE
700 ; BEGIN;
710 ; CASE CELL OF
720 ; "N": BEGIN
730 ; DRAGON(DAY-1, "W")
740 ; DRAGON(DAY-1, "N")
750 ; END;
760 ; "S": BEGIN
770 ; DRAGON(DAY-1, "E")
780 ; DRAGON(DAY-1, "S")
790 ; END;
800 ; "E": BEGIN
810 ; DRAGON(DAY-1, "E")
820 ; DRAGON(DAY-1, "N")
830 ; END;
840 ; "W": BEGIN
850 ; DRAGON(DAY-1, "W")
860 ; DRAGON(DAY-1, "S")
870 ; END;
880 ; END;
885 ; END;
890 ; END;
```



Here is a **binary tree** visualising the number of **DRAGON procedure invocations** that occur when drawing a **dragon aged four**:



Age	Calls
4	1
3	2
2	4
1	8
0	16

Each of the **tree's nodes** is labelled **C** (for **call**) and represents one **invocation**. As the **tree depth** grows, the **age** decreases and the number of **calls** grows. The **age** at **depth N+1** is one less than the **age** at **depth N**. The number of **calls** at **depth N+1** is **twice** the number of **calls** at **depth N**.

The number of **nodes** in a **binary tree** of **depth N** is  $\sum_{i=0}^n 2^i = 2^{n+1} - 1$ .

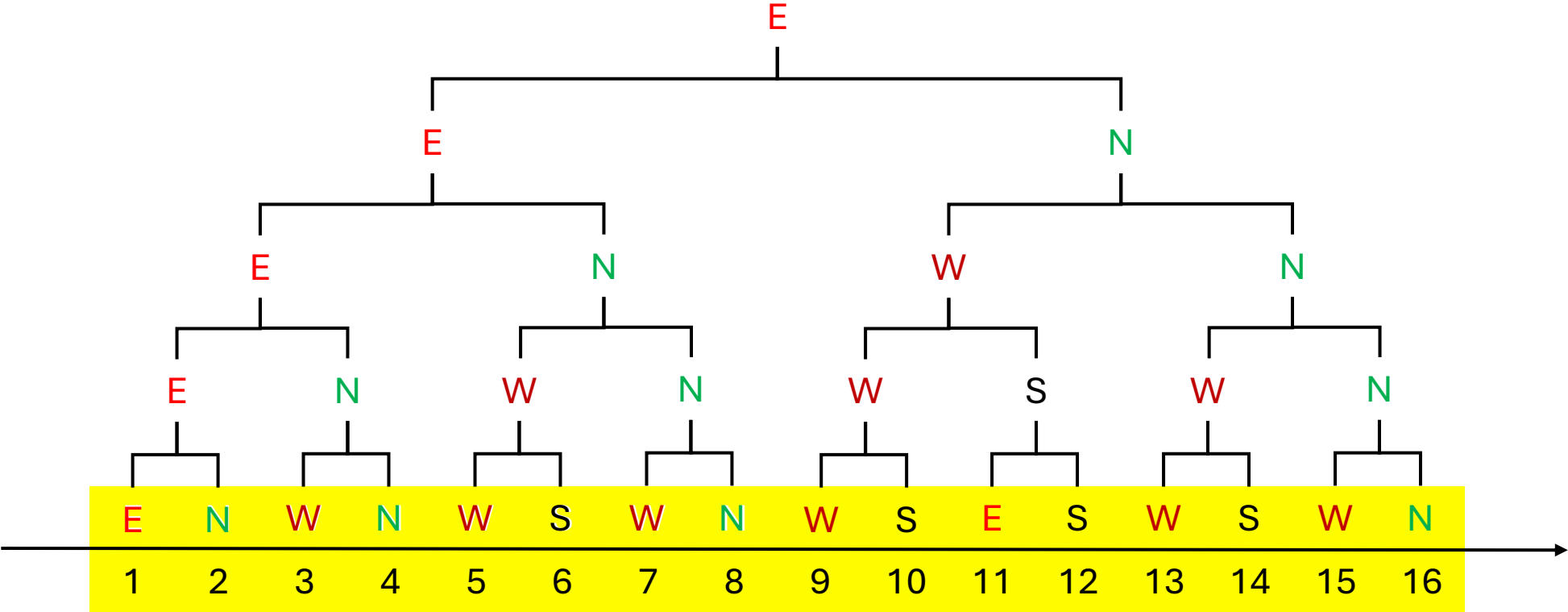
So the number of **procedure calls** for a **dragon aged four** is  $2^{4+1} - 1 = 2^5 - 1 = 32 - 1 = 31$ .

The number of **tree nodes** at **depth N** is  $2^N$ , so for a **dragon aged four**, the number of **leaf nodes** at the bottom of the **tree** is  $2^4 = 16$ .

**Leaf nodes** represent **calls** that have reached the **base case** and which therefore **invoke** the **DRAW procedure** in order to **draw** a **line**.

A **dragon aged four** is drawn by drawing 16 **lines**. The **lines** connect the 15 **points** forming the **dragon's path**.





Same **tree** as on the previous slide, but with the **leaf nodes** highlighted in yellow and assigned a **sequence number**.

Each **leaf node** represents the drawing of a **line** in the **direction** specified by the node's **label**.

The **arrow** indicates that the **lines** are drawn in **sequence** going from **left** to **right**.

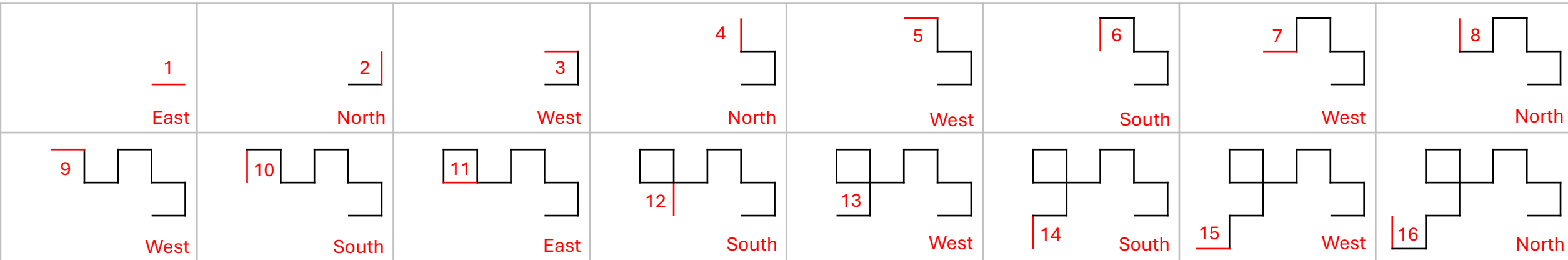
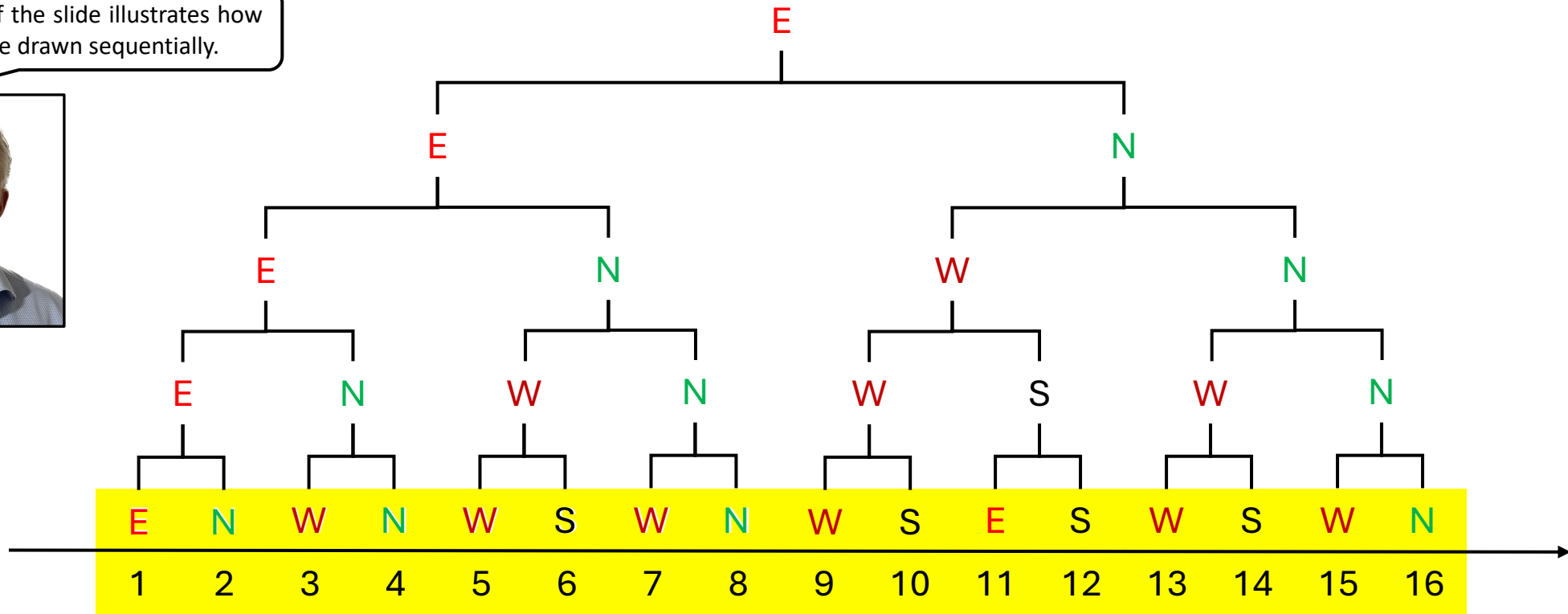
The **starting point P1** of the **first line** is **user defined**, and its **end point P2** is at **distance LENGTH eastwards** of **P1**.

The **starting point** of the **second line** is **P2** (the **end point** of the **first line**), and its **end point** is at **distance LENGTH northwards** of **P2**.

And so on.

The bottom of the slide illustrates how the 16 lines are drawn sequentially.

Age = 4  
Lines = 16





Let's start rewriting the **Pascal DRAGON procedure** in **Scala**.

```
PROCEDURE DRAGON (DAY: INTEGER; CELL: CHAR);
```

As a first step, here is a **top level Scala function** representing the **imperative shell**.

We have renamed the **day** and **cell** parameters of **DRAGON** to **age** and **direction** respectively.

The **DRAGON procedure** is **impure** because it invokes **side-effecting procedure draw**, and because it uses **global variables X1, Y1** and **LENGTH**.

The **drawDragon** function is also **impure**, because although it doesn't use any **external variables**, it also invokes a **side-effecting draw** function.

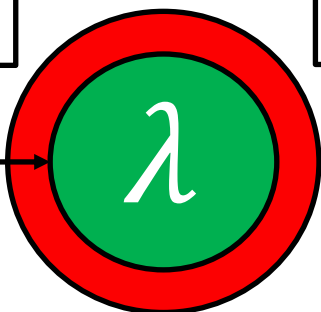
On the next slide we start looking at what **drawDragon** does.

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
  Dragon(startPoint, age, length, direction)  
    .path  
    .lines  
    .foreach(draw)
```

is the

functional  
core

imperative  
shell



Create the **dragon**.

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
    Dragon(startPoint, age, length, direction)  
        .path  
        .lines  
        .foreach(draw)
```

Ask the **dragon** for its **path**.

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
  Dragon(startPoint, age, length, direction)  
    .path  
    .lines  
    .foreach(draw)
```



The **path** of a **dragon** is computed by first creating an **initial path** containing only a **starting point**, and then **growing** the **path** according to the specified **age**, **line length** and **direction**.

```
case class Dragon(startPoint: Point, age: Int, length: Int, direction: Direction):  
  val path: DragonPath =  
    DragonPath(startPoint)  
    .grow(age, length, direction)
```

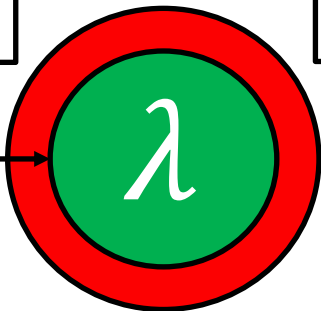
part of

Ask the **dragon** for its **path**.

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
  Dragon(startPoint, age, length, direction)  
    .path  
    .lines  
    .foreach(draw)
```

functional  
core

imperative  
shell



Ask the **path** for **lines** connecting its **points**.

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
  Dragon(startPoint, age, length, direction)  
    .path  
    .lines  
    .foreach(draw)
```



The **path** of a **dragon** is a list of **points**. We can ask a **path** for the **lines** connecting its **points**. A **line** is defined by the two **points** that it connects.

Not shown here: how to **grow** a **path**. We'll be looking at that very soon.

```
object DragonPath:
  def apply(startPoint: Point): DragonPath = List(startPoint)

  extension (path: DragonPath)
    def lines: List[Line] =
      if path.length < 2 then Nil
      else path.zip(path.tail)
```

```
type DragonPath = List[Point]

type Line = (Point, Point)

extension (line: Line)
  def start: Point = line(0)
  def end: Point = line(1)
```

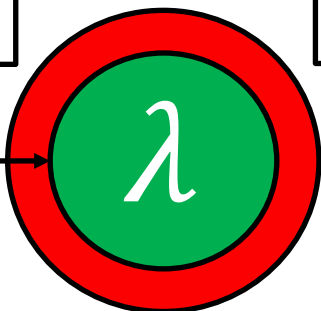
part of

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =
  Dragon(startPoint, age, length, direction)
    .path
    .lines
    .foreach(draw)
```

Ask the **path** for **lines** connecting its **points**.

functional  
core

imperative  
shell



```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
  Dragon(startPoint, age, length, direction)  
    .path  
    .lines  
    .foreach(draw)
```

For each **line**, draw the **line**.



Since we are currently rewriting the **Pascal DRAGON** procedure in **Scala**, the **draw** function, which is **side-effecting**, is **out of scope** for now, but we'll be coming back to it soon.

The next thing we have to do as part of our **rewrite** is provide a function for **growing a dragon path**, which you can see on the next slide.

```
def drawDragon(startPoint: Point, age: Int, length: Int, direction: Direction): Unit =  
  Dragon(startPoint, age, length, direction)  
    .path  
    .lines  
    .foreach(draw)
```

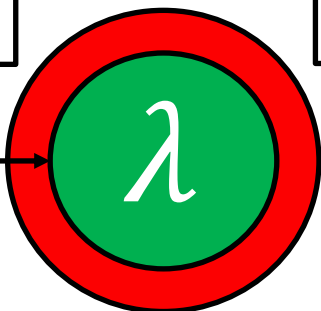
For each **line**, draw the **line**.

draw function

part of

functional  
core

imperative  
shell





The **recursive grow function** is at the **heart** of our **rewrite**.

Unlike the **DRAGON procedure**, which is **side-effecting**, in that it draws **lines** on the **screen**, the **grow function** is **pure**, because all it does is **grow** a **path** by returning a new one which has had new **points** added to its **front** (**prefixed** to it).

While the **base case** of the **DRAGON procedure** draws a **line**, the **base case** of the **grow function** **prefixes** a **path** with a new **point** computed by **translating** (moving), by the given line **length**, and in the specified **direction**, the **point** currently at the **front** of the **path**.

As for the **recursive case**, the **DRAGON procedure** invokes itself twice knowing that each invocation will **return nothing** yet result in **lines** being drawn, whereas the **grow function** invokes itself a first time to **grow** its **path** parameter, collects this **first resulting path**, and **grows** it in turn by calling itself a second time, and finally collects this **second resulting path** and returns it as its own **result**.

```
extension (path: DragonPath)
```

```
def grow(age: Int, length: Int, direction: Direction): DragonPath =
```

```
def newDirections(direction: Direction): (Direction, Direction) =
```

```
direction match
```

```
case North => (West, North)
```

```
case South => (East, South)
```

```
case East  => (East, North)
```

```
case West  => (West, South)
```

```
path.headOption.fold(path): front =>
```

```
if age == 0
```

```
then front.translate(direction, length) :: path
```

```
else
```

```
val (firstDirection, secondDirection) = newDirections(direction)
```

```
path
```

```
.grow(age - 1, length, firstDirection)
```

```
.grow(age - 1, length, secondDirection)
```

```
extension (p: Point)
```

```
def translate(direction: Direction, amount: Float)
```

```
:Point = direction match
```

```
case North => Point(p.x, p.y + amount)
```

```
case South => Point(p.x, p.y - amount)
```

```
case East  => Point(p.x + amount, p.y)
```

```
case West  => Point(p.x - amount, p.y)
```

```
540 PROCEDURE DRAGON (DAY: INTEGER; CELL: CHAR);
550 BEGIN
560 ; IF DAY=0
570 ; THEN
580 ; BEGIN
590 ; CASE CELL OF
600 ;     "N": Y2 := Y1 - LENGTH;
610 ;     "S": Y2 := Y1 + LENGTH;
620 ;     "E": X2 := X1 + LENGTH;
630 ;     "W": X2 := X1 - LENGTH;
640 ; END;
650 ; DRAW(X1, Y1, X2, Y2);
660 ; X1:= X2;
670 ; Y1:= Y2;
680 ; END;
690 ; ELSE
700 ; BEGIN;
710 ; CASE CELL OF
720 ;     "N": BEGIN
730 ;         DRAGON(DAY-1, "W")
740 ;         DRAGON(DAY-1, "N")
750 ;     END;
760 ;     "S": BEGIN
770 ;         DRAGON(DAY-1, "E")
780 ;         DRAGON(DAY-1, "S")
790 ;     END;
800 ;     "E": BEGIN
810 ;         DRAGON(DAY-1, "E")
820 ;         DRAGON(DAY-1, "N")
830 ;     END;
840 ;     "W": BEGIN
850 ;         DRAGON(DAY-1, "W")
860 ;         DRAGON(DAY-1, "S")
870 ;     END;
880 ; END;
885 ; END;
890 ; END;
```

Pascal 64 imperative version

```
def drawDragon(start: Point, age: Int, length: Int, direction: Direction): Unit =
  Dragon(start, age, length, direction)
    .path
    .lines
    .foreach(draw)
```

```
case class Dragon(start: Point, age: Int, length: Int, direction: Direction):
  val path: DragonPath =
    DragonPath(start)
    .grow(age, length, direction)
```

```
type DragonPath = List[Point]

object DragonPath:
  def apply(start: Point): DragonPath = List(start)

extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    path.headOption.fold(path): front =>
      if age == 0
      then front.translate(direction, length) :: path
      else
        val (firstDirection, secondDirection) = newDirections(direction)
        path
          .grow(age - 1, length, firstDirection)
          .grow(age - 1, length, secondDirection)

  def lines: List[Line] =
    if path.length < 2 then Nil
    else path.zip(path.tail)
```

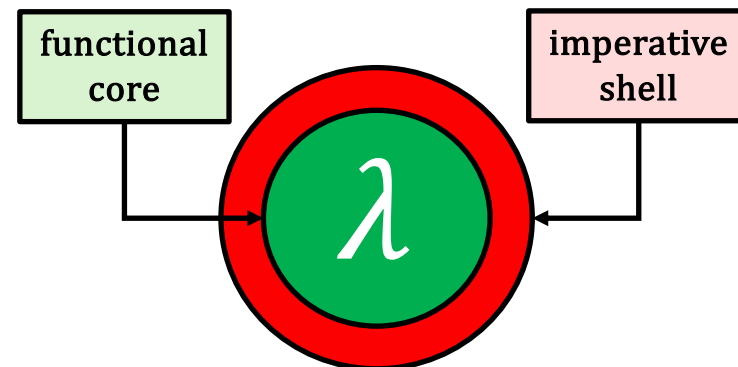
```
type Line = (Point, Point)

extension (line: Line)
  def start: Point = line(0)
  def end: Point   = line(1)
```

```
enum Direction:
  case North, East, South, West
```

```
case class Point(x: Float, y: Float)

extension (p: Point)
  def translate(direction: Direction, amount: Float)
    : Point = direction match
      case North => Point(p.x, p.y + amount)
      case South => Point(p.x, p.y - amount)
      case East  => Point(p.x + amount, p.y)
      case West  => Point(p.x - amount, p.y)
```



```
def drawDragon(start: Point, age: Int, length: Int, direction: Direction): Unit =
  Dragon(start, age, length, direction)
    .path
    .lines
    .foreach(draw)
```

```
case class Dragon(start: Point, age: Int, length: Int, direction: Direction):
  val path: DragonPath =
    DragonPath(start)
    .grow(age, length, direction)
```

```
type DragonPath = List[Point]

object DragonPath:
  def apply(start: Point): DragonPath = List(start)

extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    path.headOption.fold(path): front =>
      if age == 0
      then front.translate(direction, length) :: path
      else
        val (firstDirection, secondDirection) = newDirections(direction)
        path
          .grow(age - 1, length, firstDirection)
          .grow(age - 1, length, secondDirection)

  def lines: List[Line] =
    if path.length < 2 then Nil
    else path.zip(path.tail)
```

```
type Line = (Point, Point)
```

```
extension (line: Line)
  def start: Point = line(0)
  def end: Point = line(1)
```

```
enum Direction:
  case North, East, South, West
```

```
case class Point(x: Float, y: Float)

extension (p: Point)
  def translate(direction: Direction, amount: Float)
    : Point = direction match
      case North => Point(p.x, p.y + amount)
      case South => Point(p.x, p.y - amount)
      case East  => Point(p.x + amount, p.y)
      case West  => Point(p.x - amount, p.y)
```

```
540 PROCEDURE DRAGON (DAY: INTEGER; CELL: CHAR);
550 BEGIN
560 ; IF DAY=0
570 ; THEN
580 ; BEGIN
590 ; CASE CELL OF
600 ; "N": Y2 := Y1 - LENGTH;
610 ; "S": Y2 := Y1 + LENGTH;
620 ; "E": X2 := X1 + LENGTH;
630 ; "W": X2 := X1 - LENGTH;
640 ; END;
650 ; DRAW(X1, Y1, X2, Y2);
660 ; X1:= X2;
670 ; Y1:= Y2;
680 ; END;
690 ; ELSE
700 ; BEGIN;
710 ; CASE CELL OF
720 ; "N": BEGIN
730 ; DRAGON(DAY-1, "W")
740 ; DRAGON(DAY-1, "N")
750 ; END;
760 ; "S": BEGIN
770 ; DRAGON(DAY-1, "E")
780 ; DRAGON(DAY-1, "S")
790 ; END;
800 ; "E": BEGIN
810 ; DRAGON(DAY-1, "E")
820 ; DRAGON(DAY-1, "N")
830 ; END;
840 ; "W": BEGIN
850 ; DRAGON(DAY-1, "W")
860 ; DRAGON(DAY-1, "S")
870 ; END;
880 ; END;
885 ; END;
890 ; END;
```

original Pascal 64 imperative version



Now that we have rewritten the **imperative Pascal DRAGON** procedure as a **Scala function** consisting of an **imperative shell** and a **functional core**, let's turn to the fact that both versions of the code depend on a **side-effecting draw** function used to draw a **line**.

How can we to implement the **line drawing function** in **Scala**?

Since the function is not the focus of this deck, let's just **reuse** the **approach** taken in the deck below.

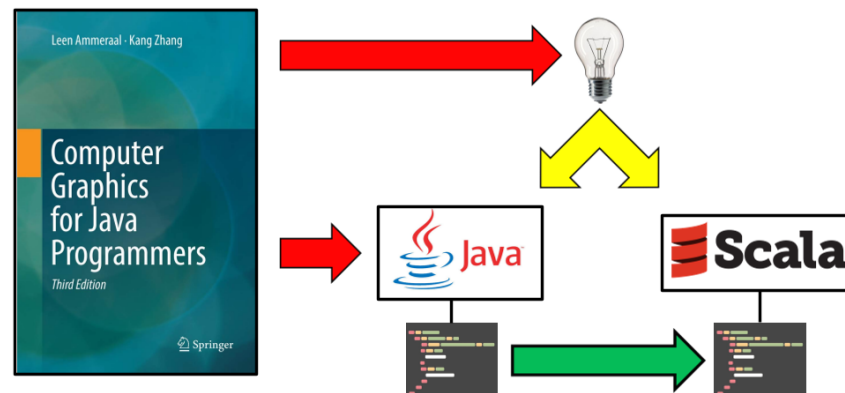
The **approach** is slightly overcomplex in that it is designed to handle **lines** whose **points** have **coordinates** that are **real numbers**, whereas in our case, **points** whose **coordinates** are **whole numbers** would suffice.

Still, the **approach** does handle for us the problem of converting **cartesian coordinates**, in which the **y coordinate** grows as a **point** moves **north**, to **screen coordinates**, in which it grows as a **point** moves **south**.

# Computer Graphics in Java and Scala

Part 1

**Continuous (Logical)** and **Discrete (Device)** Coordinates  
with a simple yet pleasing example involving concentric triangles



slides by



[@philip\\_schwarz](https://twitter.com/philip_schwarz)

[slideshare https://www.slideshare.net/pjschwarz](https://www.slideshare.net/pjschwarz)



First we extend a point with a **deviceCoords** function that maps **cartesian coordinates** to **screen coordinates**. Note that the function uses external value **panelHeight**, the height of the **graphics panel** on which the **line** is to be drawn.

Next we define a **draw** function that draws a **line** on the **screen**.

To do this, it first uses the newly introduced **deviceCoords** function to convert the **points** of the line from **cartesian coordinates** to **screen coordinates**. Note how in doing this it also depends on external value **panelHeight**.

Next, it invokes a **drawLine** function provided by external value **g**, which is a **graphics context**.

```
case class Point(x: Float, y: Float)
```

```
extension (p: Point)
```

```
def deviceCoords(panelHeight: Int): (Int, Int) =  
  (Math.round(p.x), panelHeight - Math.round(p.y))
```

```
def translate(direction: Direction, amount: Float): Point =  
  direction match  
    case North => Point(p.x, p.y + amount)  
    case South => Point(p.x, p.y - amount)  
    case East  => Point(p.x + amount, p.y)  
    case West  => Point(p.x - amount, p.y)
```

```
def draw(line: Line): Unit =  
  val (ax, ay) = line.start.deviceCoords(panelHeight)  
  val (bx, by) = line.end.deviceCoords(panelHeight)  
  g.drawLine(ax, ay, bx, by)
```



On the next slide we are now in a position to define a **graphics panel** that draws the **dragon**! It is in that **panel** that the above **draw** function lives.

```
import java.awt.{Color, Graphics}
import javax.swing.*

class DragonPanel(lineColour: Color, backgroundColour: Color) extends JPanel:
```

```
  override def paintComponent(g: Graphics): Unit =
```

```
    val panelHeight = getSize().height - 1
```

```
    def startPoint: Point =
      val panelWidth = getSize().width - 1
      val panelCentre = Point(panelWidth / 2, panelHeight / 2)
      panelCentre
        .translate(South, panelHeight / 7)
        .translate(West, panelWidth / 5)
```

```
    def draw(line: Line): Unit =
      val (ax, ay) = line.start.deviceCoords(panelHeight)
      val (bx, by) = line.end.deviceCoords(panelHeight)
      g.drawLine(ax, ay, bx, by)
```

```
    def drawDragon(start: Point, age: Int, length: Int, direction: Direction): Unit =
      Dragon(start, age, length, direction)
        .path
        .lines
        .foreach(draw)
```

```
    super.paintComponent(g)
    setBackground(backgroundColour)
    g.setColor(lineColour)
```

```
    drawDragon(startPoint, age = 17, length = 1, direction = East)
```

A **graphics panel** on which the **dragon** is to be drawn.

The **panelHeight** .value used by both **draw** and **startPoint**.

The **point** where the drawing of the **dragon** begins. Currently computed to allow an age 17 **dragon** to fit in the **panel**.

We defined this on the previous slide.

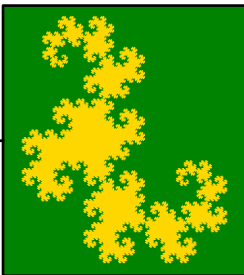
The first **Scala** function that we looked at when we started rewriting the **Pascal DRAGON** procedure.

In that context, the function constituted the whole of the imperative shell.

This **graphics Panel** that we are now adding in order to **complete** the **drawDragon function** (by adding **draw**) and to **make use of** it, becomes part of the **imperative shell**.

Boilerplate code

Should draw the **dragon** that we saw at the beginning of the **deck**.





Now that we have defined a **panel**, we need a **frame** to hold an instance of the **panel**.

Here is a **boilerplate** function to create a 600 x 600 pixel **frame**.

When we instantiate the **panel** we specify a **background colour** of **green** and a **line colour** of **gold**.

```
import java.awt.Color
import javax.swing.{JFrame, WindowConstants}

def displayDragonFrame(): Unit =
  val (gold, green) = (Color(255, 215, 0), Color(0, 128, 0))
  val panel = DragonPanel(lineColour = gold, backgroundColour = green)
  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Heighway's Dragon")
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
  frame.setSize(600,600)
  frame.add(panel)
  frame.setVisible(true)
```



All that we need now in order to finally draw a **dragon** is a **main function** that creates a **frame** using the function defined on the previous slide.

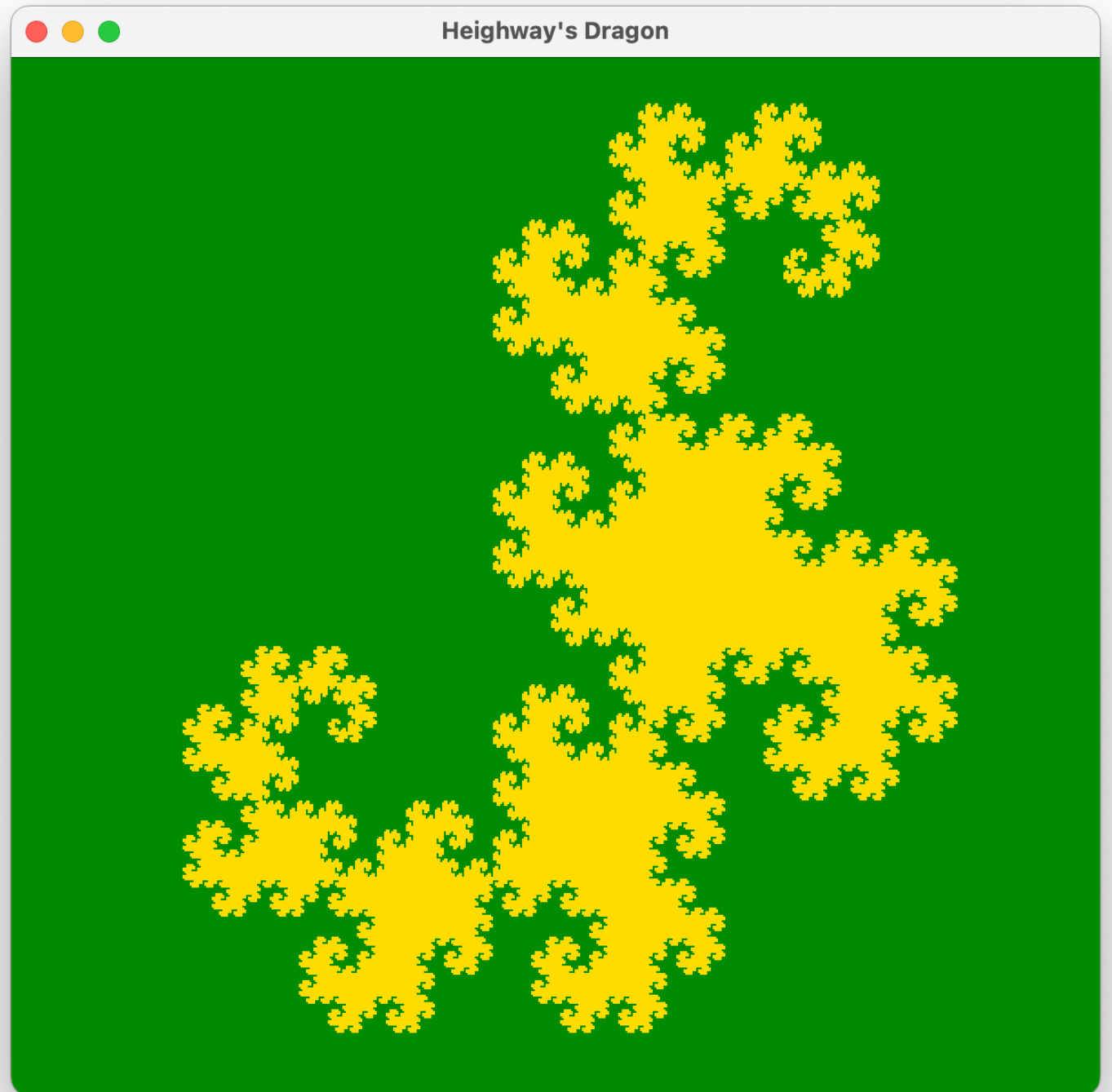
On the next slide you can see the result of **running** the **main function**.

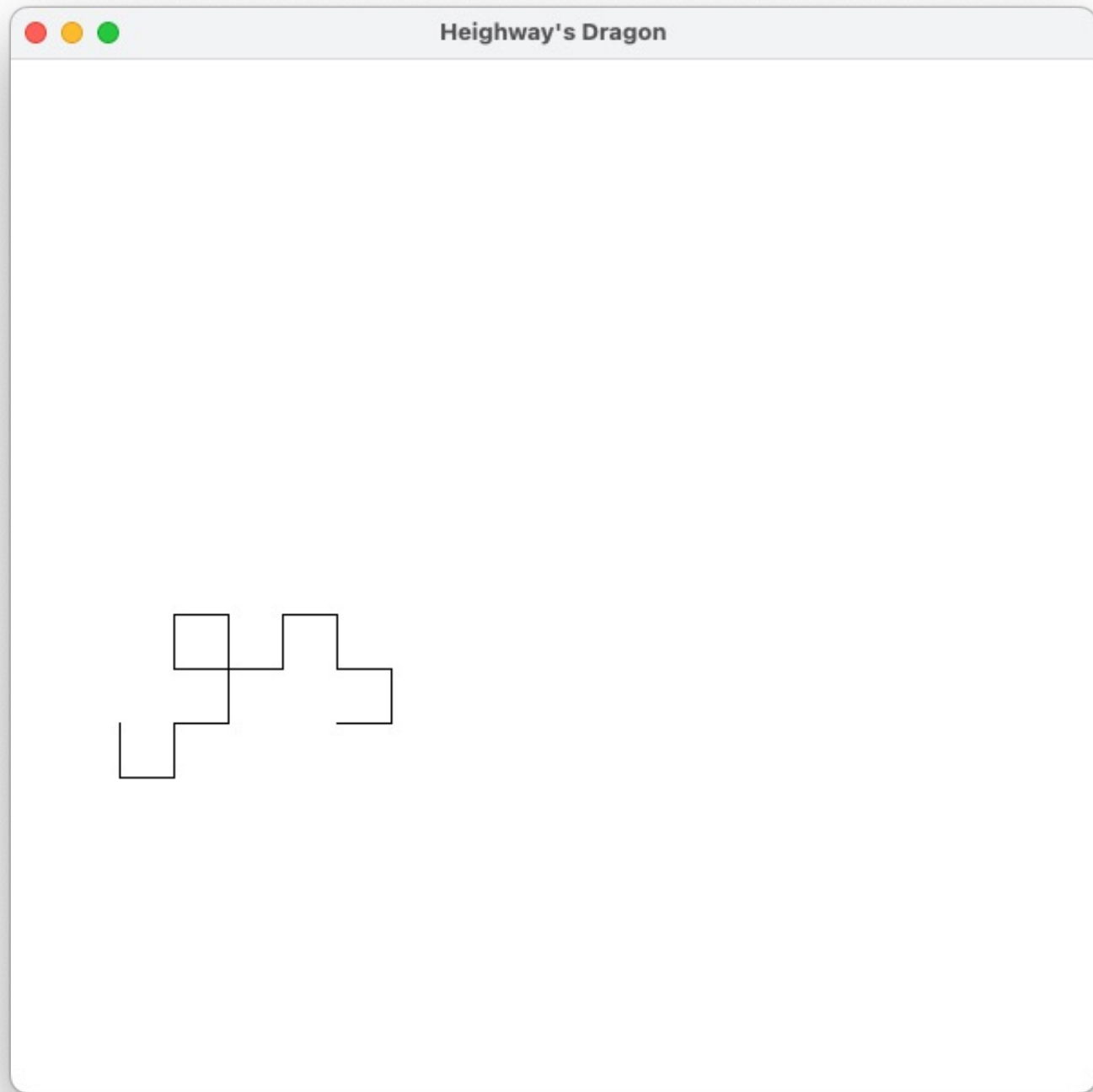
```
import javax.swing.SwingUtilities

@main def main(): Unit =
  // Create the frame/panel on the event dispatching thread.
  SwingUtilities.invokeLater(
    new Runnable():
      def run(): Unit = displayDragonFrame()
  )
```

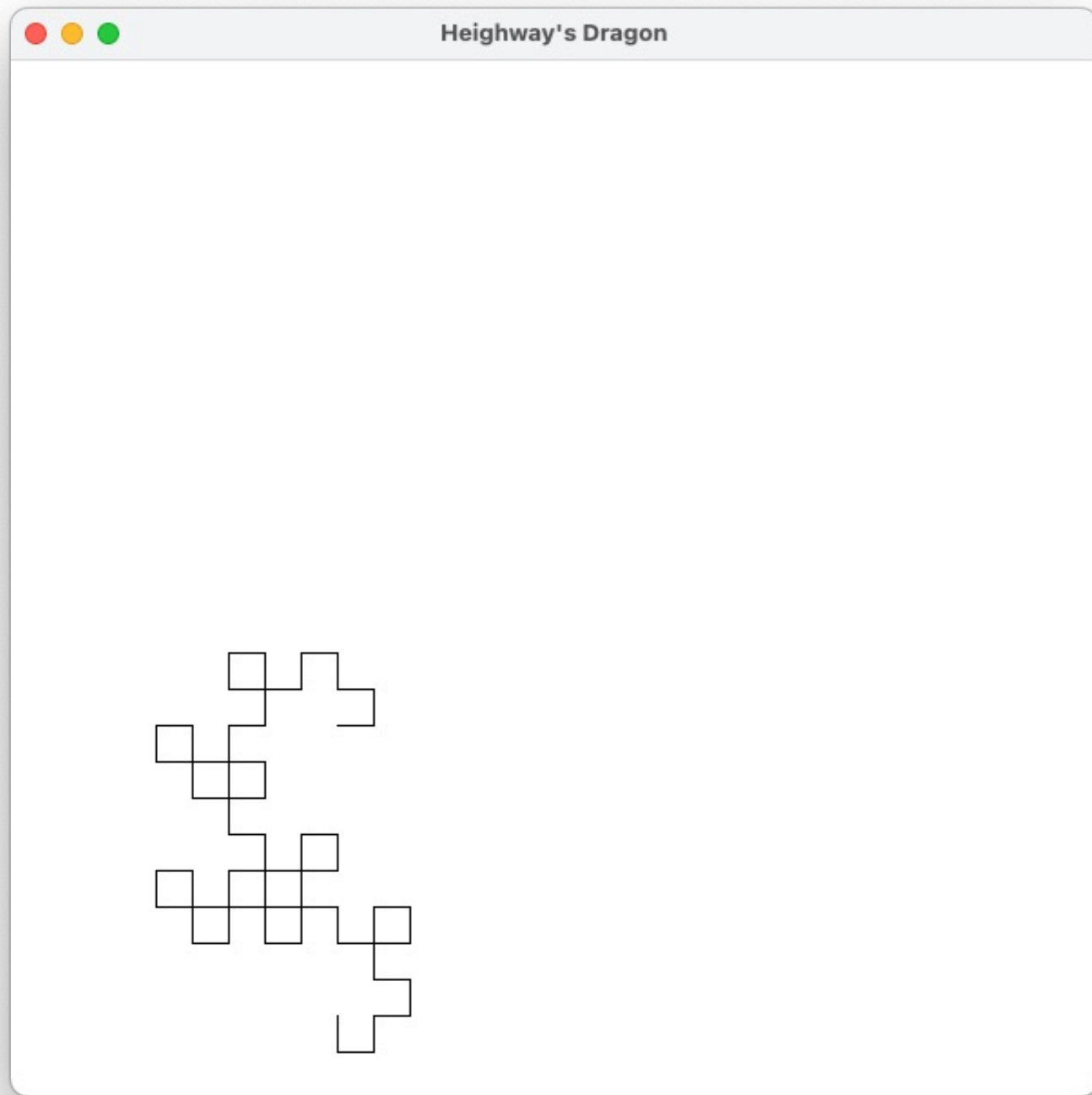
It works nicely!

In the next few slides we'll see the results of **running** the program **multiple times** with **white background** and **black line colour**, and with different **ages** and **line lengths**.

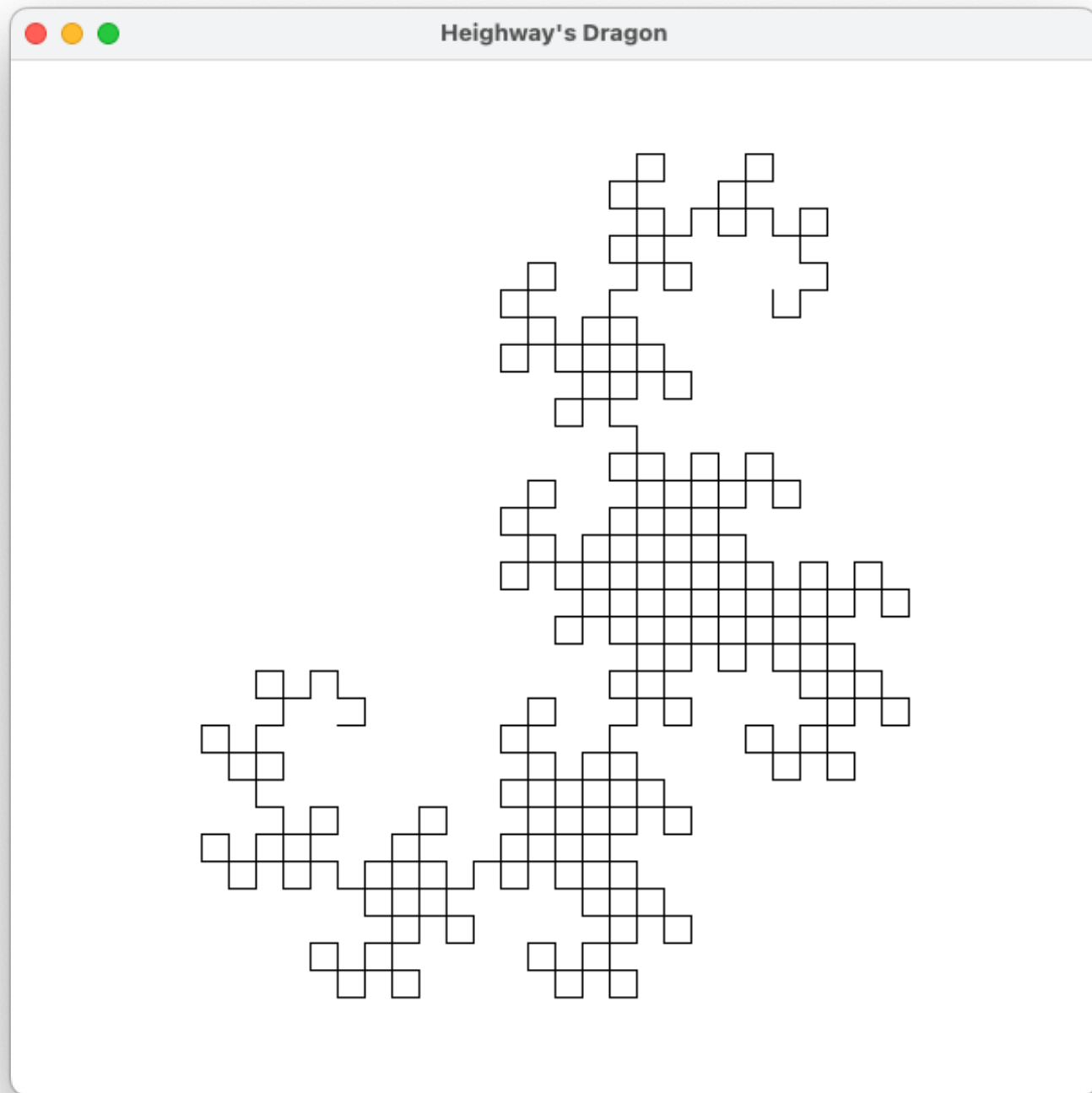




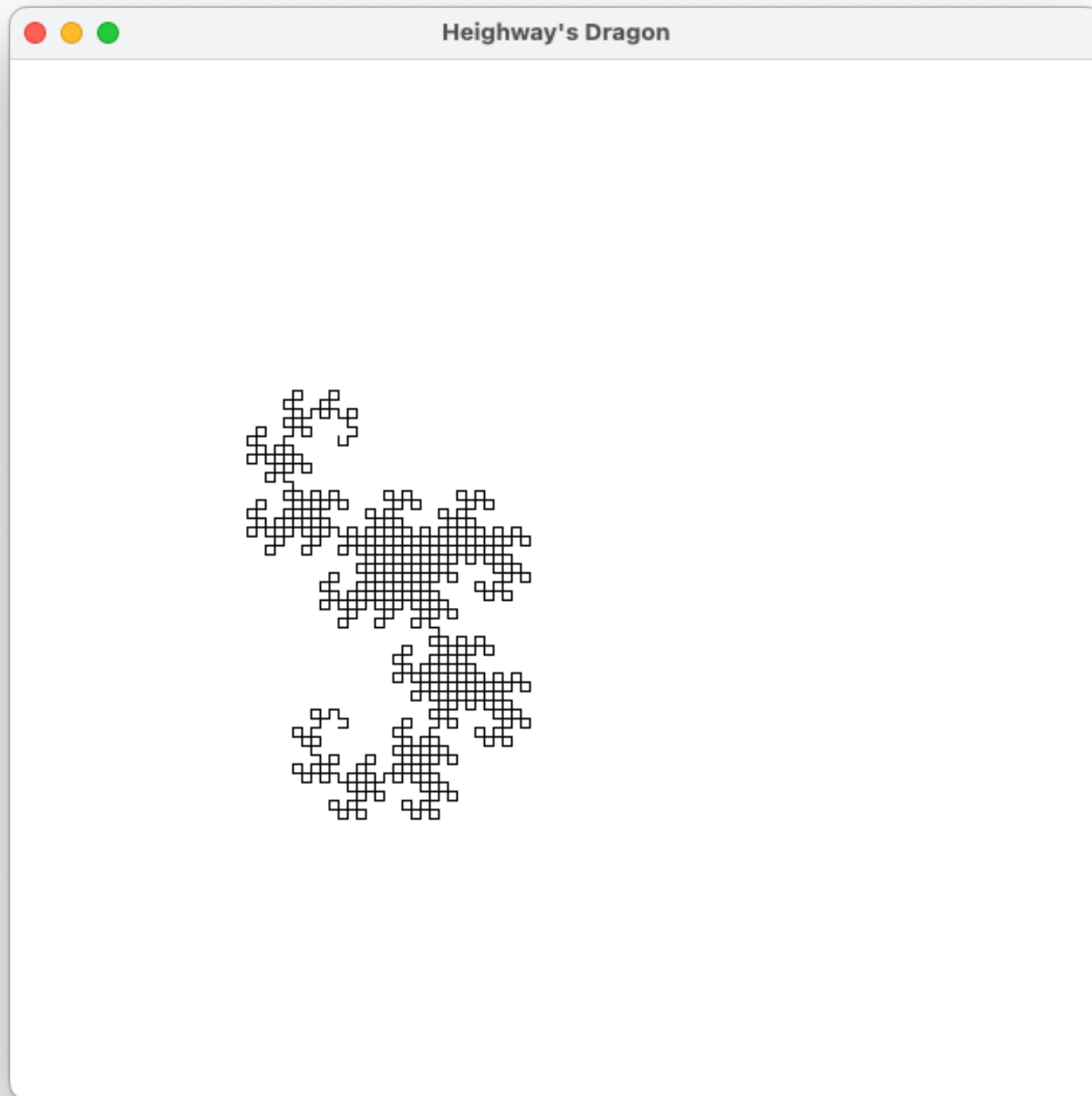
age = 4  
length = 30



age = 6  
length = 20



age = 9  
length = 15



age = 11  
length = 10



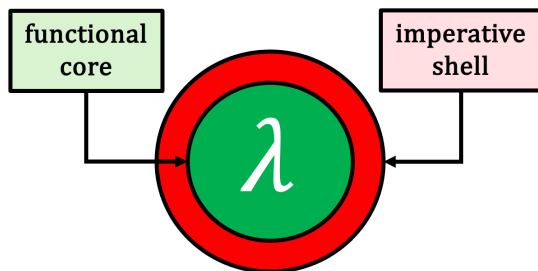
In **conclusion**, the next two slides **recap** the **code** of the **whole Scala program**.

```
import javax.swing.SwingUtilities

@main def main(): Unit =
  // Create the frame/panel on the event dispatching thread.
  SwingUtilities.invokeLater(
    new Runnable():
      def run(): Unit = displayDragonFrame()
  )
```

```
import java.awt.Color
import javax.swing.{JFrame, WindowConstants}

def displayDragonFrame(): Unit =
  val (gold, green) = (Color(255, 215, 0), Color(0, 128, 0))
  val panel = DragonPanel(lineColour = gold, backgroundColour = green)
  JFrame.setDefaultLookAndFeelDecorated(true)
  val frame = new JFrame("Heighway's Dragon")
  frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE)
  frame.setSize(600, 600)
  frame.add(panel)
  frame.setVisible(true)
```



```
import java.awt.{Color, Graphics}
import javax.swing.*

class DragonPanel(lineColour: Color, backgroundColour: Color) extends JPanel:

  override def paintComponent(g: Graphics): Unit =

    val panelHeight = getSize().height - 1

    def startPoint: Point =
      val panelWidth = getSize().width - 1
      val panelCentre = Point(panelWidth / 2, panelHeight / 2)
      panelCentre
        .translate(South, panelHeight / 7)
        .translate(West, panelWidth / 5)

    def draw(line: Line): Unit =
      val (ax, ay) = line.start.deviceCoords(panelHeight)
      val (bx, by) = line.end.deviceCoords(panelHeight)
      g.drawLine(ax, ay, bx, by)

    def drawDragon(start: Point, age: Int, length: Int, direction: Direction): Unit =
      Dragon(start, age, length, direction)
        .path
        .lines
        .foreach(draw)

    super.paintComponent(g)
    setBackground(backgroundColour)
    g.setColor(lineColour)

    drawDragon(startPoint, age = 17, length = 1, direction = East)
```

```
def drawDragon(start: Point, age: Int, length: Int, direction: Direction): Unit =
  Dragon(start, age, length, direction)
    .path
    .lines
    .foreach(draw)
```

```
case class Dragon(start: Point, age: Int, length: Int, direction: Direction):
  val path: DragonPath =
    DragonPath(start)
    .grow(age, length, direction)
```

```
type DragonPath = List[Point]

object DragonPath:
  def apply(start: Point): DragonPath = List(start)

extension (path: DragonPath)

  def grow(age: Int, length: Int, direction: Direction): DragonPath =

    def newDirections(direction: Direction): (Direction, Direction) =
      direction match
        case North => (West, North)
        case South => (East, South)
        case East  => (East, North)
        case West  => (West, South)

    path.headOption.fold(path): front =>
      if age == 0
      then front.translate(direction, length) :: path
      else
        val (firstDirection, secondDirection) = newDirections(direction)
        path
          .grow(age - 1, length, firstDirection)
          .grow(age - 1, length, secondDirection)

  def lines: List[Line] =
    if path.length < 2 then Nil
    else path.zip(path.tail)
```

```
type Line = (Point, Point)

extension (line: Line)
  def start: Point = line(0)
  def end: Point   = line(1)
```

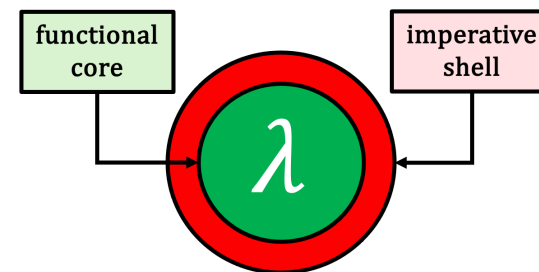
```
enum Direction:
  case North, East, South, West
```

```
case class Point(x: Float, y: Float)

extension (p: Point)

  def deviceCoords(panelHeight: Int): (Int, Int) =
    (Math.round(p.x), panelHeight - Math.round(p.y))

  def translate(direction: Direction, amount: Float): Point =
    direction match
      case North => Point(p.x, p.y + amount)
      case South => Point(p.x, p.y - amount)
      case East  => Point(p.x + amount, p.y)
      case West  => Point(p.x - amount, p.y)
```





That's all for **part 1**.

I hope you enjoyed that.

In **part 2** we'll make the **program** much **more convenient** in that it will allow us to **easily change dragon parameters** and **redraw** the **dragon** each time without having to **rerun** the **program**.

More importantly, we'll be using the **concept of rotation about a point** to **exploit** the **self-similarity** of **Heighway's dragon** and **rewrite** the **grow function** so that it is **simpler**, and so that it is therefore very **easy to understand** how it manages to compute the **path** of a **dragon**.

See you in part 2.