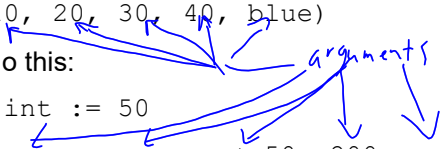
6.3 Value Parameters

Value parameters allow you to send information to a procedure. Consider the `drawline` procedure we saw at the beginning of the year. We can use it to draw a line anywhere we want in any colour we want. The procedure is not psychic. We need to tell it where to draw the line. We do that using value parameters. The `drawline` procedure has 5 parameters.

We can call it like this:

```
drawline(10, 20, 30, 40, blue)
```

or we could do this:




```
var num : int := 50
drawline(num, maxx, num + 50, 200, red)
```

In each case there needs to be 5 arguments to match up with the 5 value parameters in the procedure definition of drawline (which is part of Turing). The arguments are separated by commas.

Let's define our own procedure that uses value parameters. Turing has a procedure called `drawoval` that we can use to draw circles. It takes 5 arguments like `drawline`. You have to give both a horizontal radius and a vertical one. This is a bit redundant when you want a circle, since they'll both be the same. So let's make a procedure called `drawcircle` that will only need four arguments (x and y of the centre, the radius and the colour). We make parameters by declaring variables inside of brackets after the name of the procedure (but don't use the word `var`).

```
procedure drawcircle (x : int, y : int, r : int, col : int)
   drawoval(x, y, r, r, col)
end drawcircle
```

To draw a circle with a centre at (50, 70) with a radius of 20 in green we would say:

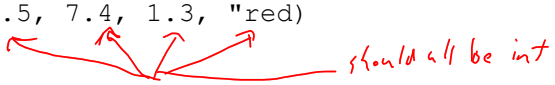
```
drawcircle(50, 70, 20, green)
```

We call it just like the predefined Turing procedures we've used all year. In this example all the arguments are constants. They can also be variables or calculations, too, as long as they give the right type (in this case they are all `int`). So we could do:

```
var x1 : int := 150
var y1 : int := 75
drawcircle(x1, y1, 60, yellow)
```

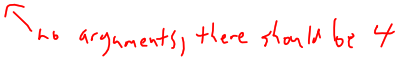
The following would be illegal because all the arguments are the wrong type, or the wrong number of arguments:

```
drawcircle (3.5, 7.4, 1.3, "red)
```




```
drawcircle (10, 20, 30, 30, green)
```


```
drawcircle
```



Parameters are like local variables since they can only be used inside a procedure. Before the procedure start however, they get initialized with the *value*s of the corresponding arguments in the procedure call. When declaring parameters we can combine ones that have the same type just like regular variable declarations. Our `drawcircle` could also be defined like this:

```
procedure drawcircle(x, y, r, col : int)
   drawoval(x, y, r, r, col)
end drawcircle
var x1 : int := 50
drawcircle (30, 40, 50, red)
drawcircle (x1, x1 + 20, 2 * x1, blue)
```

Let's trace the above program:

Global

x1 | 50

drawcircle

x | ~~30~~ 50

y | ~~40~~ 70

r | ~~50~~ 100

col | ~~red~~ blue

The first call is traced in blue

So in the procedure it does

drawoval(30, 40, 50, 50, red)

- The second call is traced in green

In the procedure it does

drawoval(50, 70, 100, blue)

Another thing to keep in mind with value parameters is that you are not allowed to change them inside the procedures. So once they are given values when you call the procedures they then act like constants.

Consider the following procedure:

```
procedure example(a : int, b : real, c: string)
   a := 55 <-- illegal, can't change the value parameter inside the procedure
   .
   .
   .
end example
```

Which of the following calls would be illegal, and why?

```
example(9, 2.5)                 X, not enough arguments
example(3, 5.5, "string")       ✓
example(5.5, 3, "word")         X first argument should be int
example(2, 5, "more")           ✓ (although 5 is an int, it can be stored in a real)
var x : int := 25
example(x, 3.1, "whatever")     ✓
example(x + 5, x / 9, "one")    ✓
```

You can also pass arrays as parameters. Suppose we wanted to print the mean of 10 numbers stored in an array. We could write a procedure like this:

```
procedure printMean(values : array 1 .. 10 of int)
   var sum : int := 0
   for i : 1 .. 10
      sum := sum + values(i)
   end for
   put "The mean is ", sum / 10
end printMean
```

The only problem is if we have a different size array (say with 500 elements) we would have to write another procedure and chance all the 10's to 500's. If we have a third size we'd need to write a third procedure. Fortunately we can make the procedure handle any size array. Here's how (the changes are in red):

```
procedure printMean(values : array 1 .. * of int)
   var sum : int := 0
   for i : 1 .. upper(values)
      sum := sum + values(i)
   end for
   put "The mean is ", sum / upper(values)
end printMean
```

The asterisk can be used to represent any size array. In the procedure we can use the upper function with the name of the array in brackets (the argument) to find out what the actual size of the array is.

Read section 6.3 in the textbook. Do exercise 6.3 #1-7 (I'll post solutions on Friday).

We will do 6.4 this coming Monday.