Homework 1

The purpose of this assignment is to warm-up with Scala.

Try to make your code as concise and clear as possible. Challenge yourself to find the most crisp, concise way of expressing the intended computation. This may mean using ways of expressing computation currently unfamilar to you.

Finally, make sure that your source file compiles and runs (using Scala 3.3.0). A solution that does not compile will *not* be graded.

For submission instructions and the due date, please see the 'README.md' file.

Problem 1 Scala Basics: Binding and Scope (12 Points)

Briefly explain for each of the following uses of names where that name is bound. Include the line number of the binding and explain your reasoning (in no more than 1-2 sentences).

(a) Consider the following Scala code.

```
1  val pi = 3.14
2  def circumference(r: Double): Double =
3    val pi = 3.14159
4    2.0 * pi * r
5
6  def area(r: Double): Double =
7    pi * r * r
```

The use of `pi` at line 4 is bound at which line? The use of `pi` at line 7 is bound at which line? **(4 Points)**

(b) Consider the following Scala code:

```
1  val x = 3
2  def f(x: Int): Int =
3     x match
4       case 0 => 0
5       case x =>
6         val y = x + 1
7         {
8           val x = y + 1
9           y
10        } * f(x - 1)
11 val y = x + f(x)
```

The use of `x` at line 3 is bound at which line? The use of `x` at line 6 is bound at which line? The use of `x` at line 10 is bound at which line? The uses of `x` at line 11 are bound at which line? **(8 Points)**

Problem 2 Scala Basics: Execution Traces (10 Points)

Consider the following recursive Scala function, which computes x^n for positive values of n:

```
def pow(x: Int, n: Int): Int =
  if n > 0 then x * pow(x, n - 1) else 1
```

(a) Show the full execution trace for the evaluation of the expression `pow(2,3)`. See Section 1.3.1 of the course notes for an example of an execution trace. **(4 Points)**

(b) Write a tail-recursive version of `pow`

```
def powTail(x: Int, n: Int): Int
```

A call `powTail(x, n)` should return the same value as `pow(x,n)`, except that it should run in constant space. Also, your implementation of `powTail` should require at most linear time in `n`. You will need to introduce a tail-recursive helper function. If you want to challenge yourself, you can aim for an implementation that achieves logarithmic time in `n`. However, this is not a requirement.

Show the execution trace for the evaluation of the call `powTail(2,3)` based on your implementation. **(6 Points)**

Problem 3 Run-Time Library (18 Points)

When we talk about language interpreters we distinguish between the *object language* and the *meta language*. The object language is the language that the input programs to the interpreter are written in. The *meta language* is the language in which we implement the interpreter itself.

Most languages come with a standard library with support for things like data structures, mathematical operators, string processing, etc. Standard library functions may be implemented in the object language (perhaps for portability) or the meta language (perhaps for implementation efficiency).

For this exercise, we will implement some library functions in Scala, our meta language, that we can imagine will be part of the run-time for our object language interpreter. In actuality, the main purpose of this exercise is to warm-up with Scala.

Please note that we do not allow the use of existing Scala library functions in this exercise, beyond primitive functions and operators on the relative data types like arithmetic operations and basic string-manipulating functions.

(a) Write a function `abs`

```
def abs(n: Double): Double
```

that returns the absolute value of `n`. This is a function that takes a value of type `Double` and returns a value of type `Double`. **(1 Points)**

(b) Write a recursive function `ar`

```
def ar(n: Int): Int
```

that returns the *arity* of `n`. The arity of `n` is the number of characters in `n`'s decimal representation (including the `-` character if the number is negative). For example, `ar(123)` returns `3` and `ar(-1)` returns `2`. One solution is to convert `n` to a `String` and then return its length. This solution is **not** allowed. Instead, implement `abs` using recursion. **(3 Points)**.

(c) Write a recursive function `rep`

```
def rep(s: String, t: String, n: Int): String
```

where `rep(s, t, n)` returns a string with `n` copies of `s` concatenated together where each consecutive pair of copies is separated by a copy of `t`. For example, the call `rep("a", ",␣", 3)` returns `"a,␣a,␣a"`. Your implementation should be tail-recursive. **(4 Points)**

(d) In this exercise, we will implement a function that approximates cube roots. To do so, we will use Newton's method (also known as Newton-Raphson). Recall from Calculus that a root of a differentiable function can be iteratively approximated by following tangent lines. More precisely, let f be a differentiable function, and let x_0 be an initial guess for a root of f . Then, Newton's method specifies a sequence of approximations $x_0, x_1, \ldots$ with the following recursive equation:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The cube root of a real number c, written $\sqrt[3]{c}$, is a real number x such that $x^3 = c$. Thus, to compute the cube root of a number c, we want to find the root of the polynomial:

$$f(x) = x^3 - c$$

By instantiating Newton's method, we obtain the following recursive equation defining a sequence of approximations for $\sqrt[3]{c}$:

$$x_{n+1} = x_n - \frac{x_n^3 - c}{3x_n^2} \quad .$$

(i) First, implement a function `approx`

```
def approx(c: Double, xn: Double): Double
```

that takes one step of approximation in computing $\sqrt[3]{c}$ (i.e., computes x_{n+1} from x_n using the above equation). **(2 Points)**

(ii) Next, implement a function `approxN`

```
def approxN(c: Double, x0: Double, n: Int): Double
```

that computes the nth approximation x_n from an initial guess x_0. You will want to call the function `approx` that you implemented in the previous part. Implement `approxN` using tail-recursion and no mutable variables (i.e., **`var`**s).**(4 Points)**

(iii) Now, implement a function `approxErr`

```
def approxErr(c: Double, x0: Double, epsilon: Double): Double
```

that is very similar to `approxN` but instead computes approximations x_n until the approximation error is within ε (epsilon), that is,

$$|x - c/x^2| < \varepsilon \ .$$

You can use your absolute value function `abs` implemented in a previous part. A wrapper function `root` is given in the template that simplifies `approxErr` calls with a choice of x_0 and ε. Again, implement `approxErr` using tail-recursion and without using mutable variables. **(4 Points)**