Tail recursion

The familiar Fibonacci sequence \( F(n) \) is given by the inductive definition:

\[
\begin{align*}
F(0) &= 1 \\
F(1) &= 1 \\
F(n) &= F(n-2) + F(n-1)
\end{align*}
\]

This definition leads immediately to a straightforward algorithm for computing \( F(n) \). This algorithm can be implemented in Scala as follows:

```scala
def fib(n: Int): Int = if (n <= 1) 1 else fib(n-1) + fib(n-2)
```

While simple, the algorithm runs in exponential time. A linear algorithm takes advantage of the fact that \( \text{fib}(n-1) \) calls \( \text{fib}(n-2) \), and reuses the result when computing \( \text{fib}(n) \).

(a) Write a recursive Scala function with the following signature:

```scala
def fib2(n: Int): (Int, Int)
```

that computes both \( \text{fib}(n) \) and \( \text{fib}(n-1) \) in linear time. The function should evaluate as follows:

\[
\begin{align*}
\text{fib2}(0) &= (1,1) \\
\text{fib2}(1) &= (1,1) \\
\text{fib2}(2) &= (2,1) \\
\text{fib2}(3) &= (3,2) \\
\text{fib2}(4) &= (5,3) \\
\ldots
\end{align*}
\]

The \( \text{fib} \) function can then be implemented efficiently using \( \text{fib2} \) and selecting the first element of the pair returned:

```scala
def fib(n: Int) = fib2(n)._1
```

(b) Try running \( \text{fib}(100000) \) in the Scala interpreter. What happens? Is the function \( \text{fib2} \) you wrote in part (a) tail recursive? Explain why or why not. If not already tail recursive, write a new version of \( \text{fib2} \), called \( \text{fib3} \), that is tail recursive. \( \text{fib3} \) might possibly with a different signature than \( \text{fib2} \) (i.e., it might have more formal parameters). Show how to implement \( \text{fib} \) using \( \text{fib3} \).
2 Scoping

In this problem we’ll look at a few variations of a simple program. Scala syntax is used, but the semantics are different: these programs might not compile or run in the Scala interpreter, and if they do, they might produce different results than what the question is asking about. Consider the following program:

```scala
def f = {
  val a = 1
  (x: Int) => x + a
}
val a = 20
println(f(300))
```

What does the program print if static scoping semantics are used? What about dynamic scoping semantics?

Now, consider this slightly modified version of the program:

```scala
def f = {
  (x: Int) => x + a
}
val a = 20
print(f(300))
```

What happens with static scoping? Dynamic scoping? Does it make a difference if errors are caught at compile time versus at run time?

Finally, consider this version:

```scala
def f = {
  val a = 1
  (x: Int) => x + a
}
print(f(300))
```

What happens with static scoping? Dynamic scoping? Again, does it make a difference if errors are caught at compile time versus at run time?