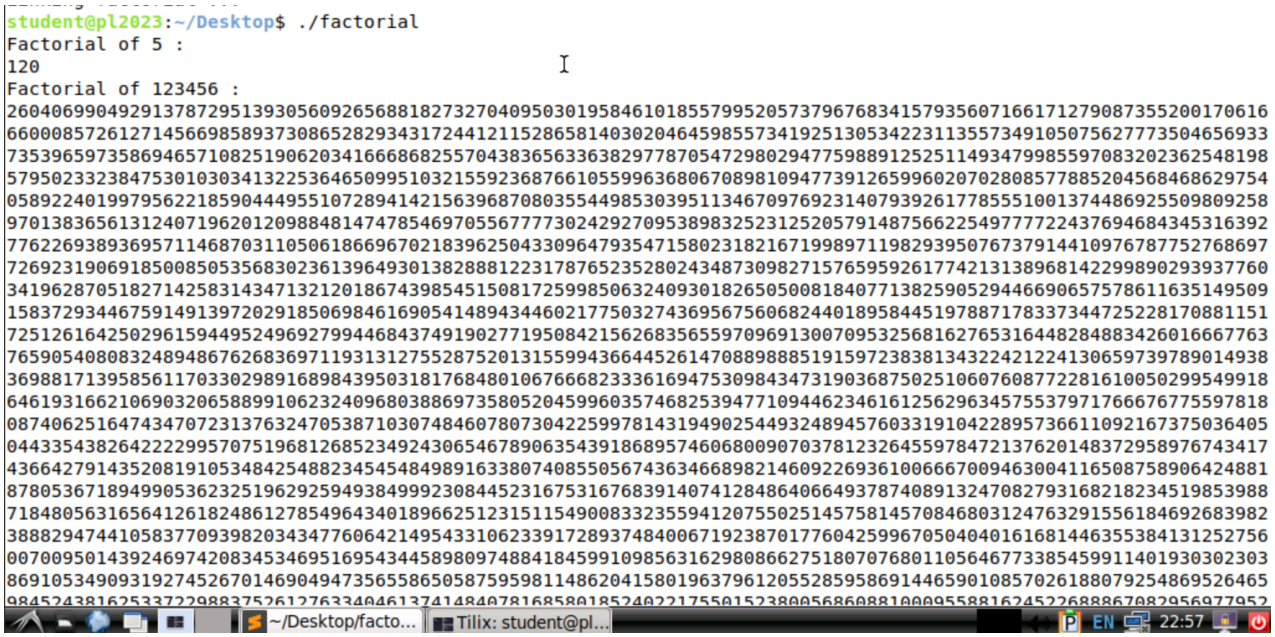
Sai Nishanth Mettu (sm11326) – Assignment 5

1) Factorial Function – Non-Tail Recursive

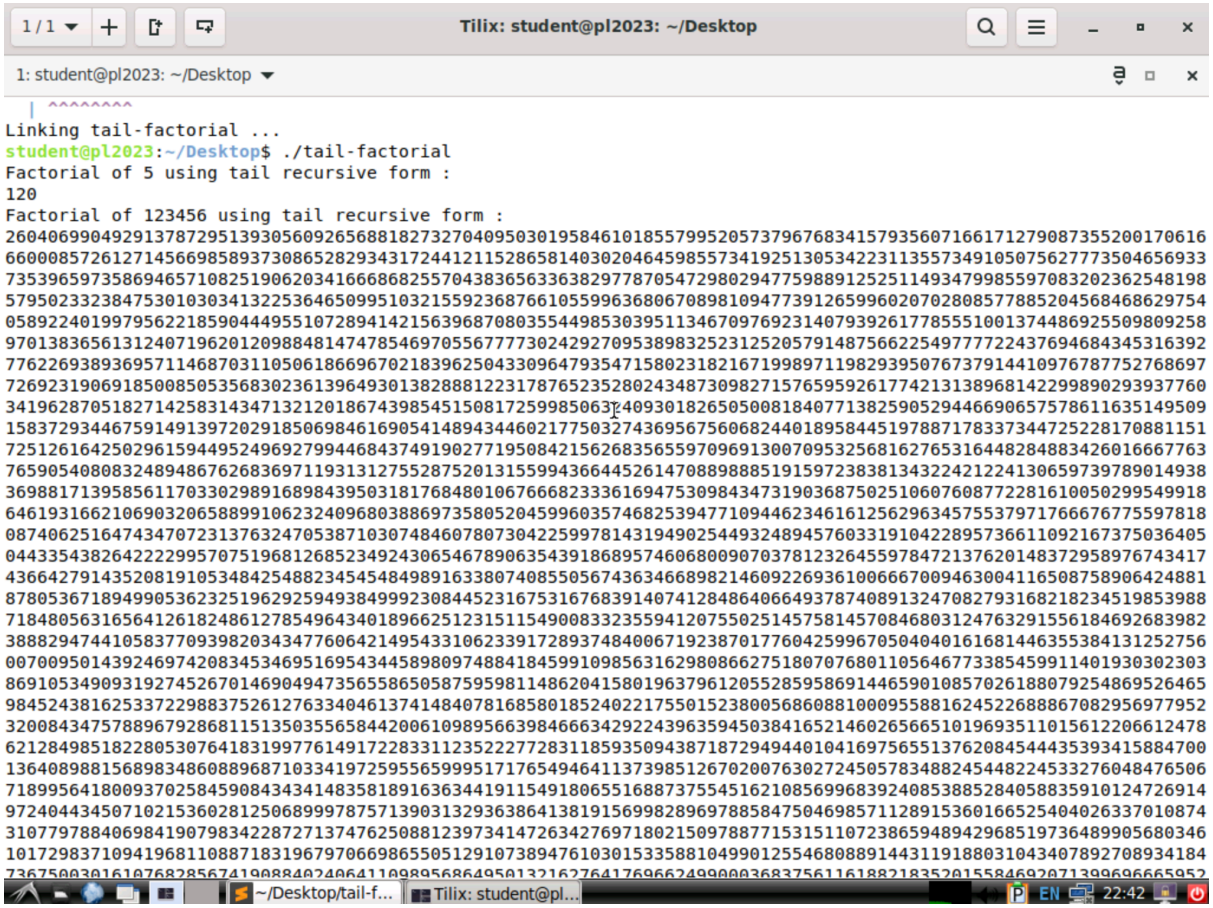


Explanation (Why Non-Tail Recursive Factorial is still able to handle large inputs)

Factorial of non-tail recursive implementation O(n) can handle fairly large inputs without encountering stack overflow errors because **of Haskell's laziness and garbage collection mechanisms.** In Haskell, even though the non-tail recursive factorial function accumulates stack frames, those frames are lazily evaluated and garbage collected when no longer needed. This means that the memory usage remains bounded, allowing the function to compute factorials of large numbers without running into memory issues.

When you call **factorial 123456**, Haskell's lazy evaluation and garbage collection mechanisms allow it to compute the result without encountering stack overflow errors.

2) Factorial Function – Tail Recursive



Explanation (Does Haskell Support Tail Recursion for Factorial and in General?)

Yes, In Haskell, tail recursion is supported, but it doesn't automatically “optimize” tail-recursive functions. Tail recursion means that the recursive call is the last operation in the function body, which can improve memory usage compared to non-tail-recursive functions. However, Haskell's runtime system doesn't guarantee automatic tail call optimization. GHC offers some tail call optimizations under certain conditions, known as "tail call optimization under certain circumstances". These optimizations can improve performance by optimizing tail calls in specific situations.

Overall, while Haskell supports Tail Recursion, it doesn't provide automatic tail call optimization, its lazy evaluation and garbage collection mechanisms help facilitate efficient execution of tail-recursive functions for many practical scenarios.

3) Fibonacci Function – Non Tail Recursive

```
Linking fibonacci ...
student@pl2023:~/Desktop$ ./fibonacci
Fibonacci of 10 using normal fibonacci recursion :
55
Fibonacci of 123456 using normal fibonacci recursion :
```

tutorialspoint | Online Haskell Compiler

Compilers | Project | Edit

Execute | Beautify | Share | Source Code

```haskell
fibonacci :: Integer -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) +
    fibonacci (n - 2)

main = do
    putStrLn "Fibonacci of 10 using: "
    print $ fibonacci 10
    putStrLn "Fibonacci of 20 using: "
    print $ fibonacci 20
    putStrLn "Fibonacci of 30 using: "
    print $ fibonacci 30
    putStrLn "Fibonacci of 40 using: "
    print $ fibonacci 40
    putStrLn "Fibonacci of 50 using: "
    print $ fibonacci 50
    putStrLn "Fibonacci of 60 using: "
    print $ fibonacci 60
    putStrLn "Fibonacci of 123456 : "
    print $ fibonacci 123456
```

Terminal

```
[1 of 1] Compiling Main             (
    main.hs, main.o )
main.hs:7:1: warning: [-Wtabs]
    Tab character found here, and in 13
        further locations.
    Please use spaces instead.
  |
7 |         putStrLn "Fibonacci of 10
    using: "
  | ^^^^^^^^
Linking main ...
Fibonacci of 10 using:
55
Fibonacci of 20 using:
6765
Fibonacci of 30 using:
832040
Fibonacci of 40 using:
102334155
Fibonacci of 50 using:
```

Why Non-Tail Recursion for Fibonacci eats up Memory, executes extremely slowly and times out eventually and why there isn’t any error message, which we saw in Lisp?

The non-tail recursive implementation of the Fibonacci function in Haskell becomes increasingly slow for larger inputs due to its exponential time complexity. It is **O(2^n)...** This inefficiency arises because the function recalculates the Fibonacci numbers for smaller values multiple times, leading to redundant computations.

In a non-tail recursive Fibonacci implementation like the one you provided, the recursive calls branch out exponentially. For each value of n, two recursive calls are made to calculate fib (n - 1) and fib (n - 2). As n increases, the number of redundant computations grows rapidly, resulting in a significant slowdown in execution time, directly as the result of no optimization.

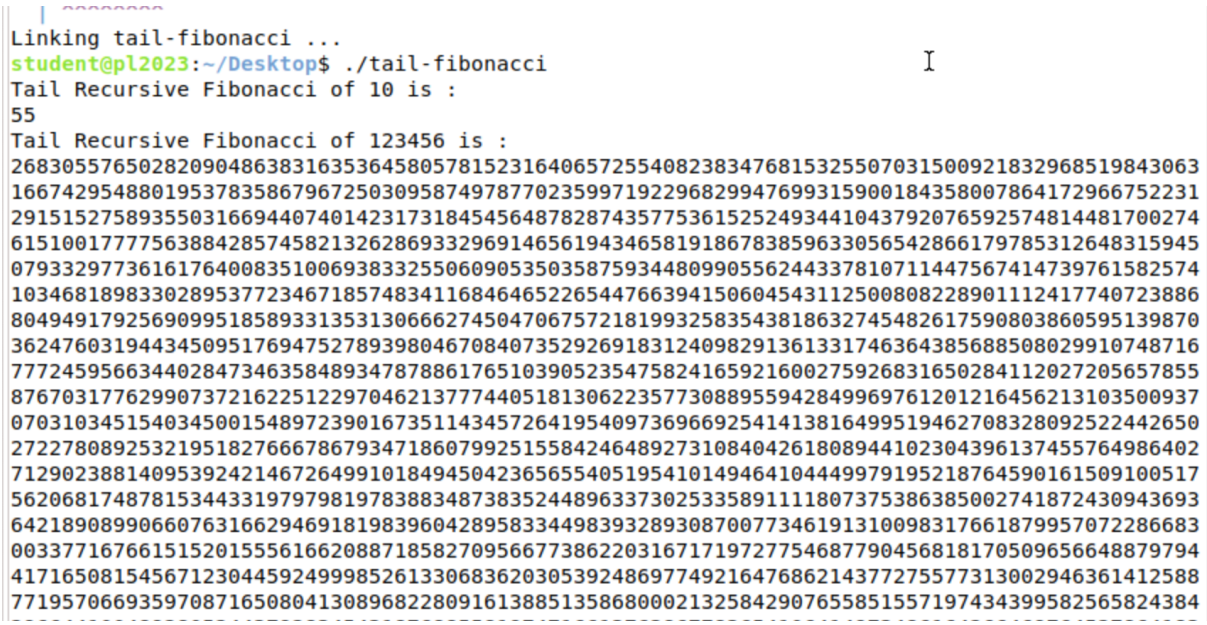
Despite the inefficiency, the non-tail recursive Fibonacci function doesn't throw errors because it's still valid Haskell code. It will eventually compute the Fibonacci numbers correctly for reasonably small inputs, although it may take a long time for larger inputs due to the exponential time complexity. While GHC Compiler does optimize (by Lazy Evaluation and Garbage Collection to a certain degree by itself, as seen in the earlier Non-Tail Factorial Problem.. it still isn’t fast enough for exponential issues like O(2^n).

→ Why Non Tail Recursive Factorial worked for larger inputs but not Fibonacci? (why is Fibonacci so slow)

The reason why non-tail recursive factorial worked for larger inputs but not Fibonacci is because the factorial function's time complexity grows linearly with the input size. **O(n)....**In a non-tail recursive factorial implementation, each recursive call multiplies the current result by the next number until it reaches the base case. This linear growth allows the function to handle larger inputs without encountering stack overflow errors, although it may still suffer from performance issues for very large inputs due to the redundant calculations. However, the **exponential growth O(2^n)** of the Fibonacci function's time complexity in the non-tail recursive implementation causes it to become extremely slow for relatively small inputs like 50, making it impractical for larger inputs.

4) Fibonacci Function – Tail Recursive

```
Linking tail-fibonacci ...
student@pl2023:~/Desktop$ ./tail-fibonacci
Tail Recursive Fibonacci of 10 is :
55
Tail Recursive Fibonacci of 123456 is :
```



Explanation :

1. The recursive call **sub (n - 1) b (a + b)** is in the tail position, meaning it's the last operation performed in each recursive step. This allows the Haskell compiler GHC or GHCI to optimize the recursion by reusing the same stack frame for each recursive call, instead of creating a new stack frame for each call. This optimization prevents stack overflow errors for large inputs, allowing the function to compute Fibonacci numbers efficiently even for very large inputs.

The optimization process works as follows:

- When the compiler encounters a tail-recursive function like **fibonacciTail**, it recognizes that the recursive call is in the tail position and can be optimized.
- The compiler transforms the tail-recursive function into an equivalent iterative loop internally, removing the need to create a new stack frame for each recursive call.
- This optimization, known as tail call optimization (TCO) or tail call elimination (TCE), eliminates the risk of stack overflow errors for large inputs by reusing the same stack frame for each recursive call.

Overall, in Summary!... the tail-recursive Fibonacci function in Haskell is optimized by leveraging tail call optimization, which transforms the recursion into an iterative process, preventing stack overflow errors and allowing efficient computation of Fibonacci numbers even for very large inputs.

FINAL INFERENCE & KNOWLEDGE GAIN

Yes, tail recursion is supported in Haskell.

Haskell compilers like GHC (Glasgow Haskell Compiler) support tail recursion optimization (TCO) “only under certain conditions”. When a function meets the criteria for tail recursion, GHC is able to apply TCO, which effectively transforms the recursive function into an iterative one, thus avoiding the accumulation of stack frames and preventing stack overflow errors.

Under the right circumstances, GHC can optimize tail-recursive functions to execute efficiently even for large inputs, making tail recursion a viable and supported technique in Haskell for writing efficient and scalable code.

The criteria for tail recursion, which enables TCO, include:

1. **Tail Call**: The recursive call must be the last operation in the function body. In other words, there should be no further computation to be done after the recursive call.
2. **No Accumulation of Deferred Operations**: There should be no accumulation of deferred operations after the recursive call. Deferred operations include pending arithmetic operations or further function calls.
3. **Tail Call in All Branches**: In functions with multiple recursive branches (e.g., pattern matching), each branch should end with a tail call. This ensures that TCO can be applied to all branches of the recursion.

When these criteria are met, the compiler can optimize tail-recursive functions by reusing the same stack frame for each recursive call, effectively transforming the recursion into an iterative process. This optimization prevents the accumulation of stack frames, leading to improved memory usage and prevention of stack overflow errors for large inputs.