FUNCTIONAL PROGRAMMING

MOISE GABRIEL

① Give a fold for the type Tree a ::= Node (Tree a) (Tree a) | Leaf a. Use it to implement a non-recursive function of type Tree a -> [a] that returns the leaves. The order is not important, however the complexity should be O(n).

```
> data Tree a = Node (Tree a) (Tree a) | Leaf a

> foldTree :: (b -> b -> b) -> (a -> b) -> Tree a -> b
> foldTree node leaf = f
>         where f (Leaf x) = leaf x
>               f (Node l r) = node (f l) (f r)
```

The function flatten, given an argument of type Tree a, returns its leaves in a list, but its complexity is quadratic in the worst-case scenario:

```
> flatten :: Tree a -> [a]
> flatten (Leaf x) = [x]
> flatten (Node l r) = flatten l ++ flatten r
```

To make it linear, we will define a function flatCat such that flatCat ys t = flatten t ++ ys for all trees t.

I flatCat ys (Leaf x) =
= {definition of flatCat}
flatten (Leaf x) ++ ys =
= {definition of flatten}
[x] ++ ys =
= {definition of (++)}
x : ys

II flatCat ys (Node l r) =
= {definition of flatCat}
flatten (Node l r) ++ ys =
= {definition of flatten}
(flatten l ++ flatten r) ++ ys =
= {associativity of (++)}
flatten l ++ (flatten r ++ ys) =
= {definition of flatCat}
flatCat (flatCat ys r) l

```
> flatCat :: [a] -> Tree a -> [a]
> flatCat ys (Leaf x) = x : ys
> flatCat ys (Node l r) = flatCat (flatCat ys r) l
```

So, the linear-time definition for flatten is

```
> flatten' :: Tree a -> [a]
> flatten' = flatCat []
```

② Implement the Knapsack problem in Haskell. Your solution should have the usual complexity of O(mW). Extra logs in the complexity are only forgivable if you don't spend them to simulate arrays. One way would be to aim for something like solve :: int -> [(int, int)] -> [(int, int)]. The result represents pairs of weight/profit that can be achieved by choosing subsets of the input.

```
> Knapsack :: Int -> [(Int, Int)] -> [(Int, Int)]
> Knapsack w xs = snd $ dp w xs ((0, []) : (repeat (minBound, [])))
                        ↙ previous dynamic line = [(sum, objects chosen)]
> dp :: Int -> [(Int, Int)] -> [(Int, [(Int, Int)])] -> (Int, [(Int, Int)])
> dp w [] prev = list_max $ take (w+1) prev   -- no objects left to add
> dp w ((weight, value) : xs) prev = dp w xs (zipWith value_max prev new)   the first "weight" stay
>     where new = (take weight (repeat (minBound, []))) ++ (map add_item prev)   the same, we might add object to the others
>           add_item (x, xs) = (x + value, (weight, value) : xs)

> value_max :: (Int, [(Int, Int)]) -> (Int, [(Int, Int)]) -> (Int, [(Int, Int)])
> value_max (x, xs) (y, ys) = if (x>y) then (x, xs) else (y, ys)

> list_max :: [(Int, [(Int, Int)])] -> (Int, [(Int, Int)])
> list_max = foldr value_max (minBound, [])
```

This program was made on the computer because it seemed too hard for me to do it by hand, especially without (!!), to make it efficient.

③ Give the function of type a -> b. Do you know its usual name? How could you use it to define bottom?

If the question does not say anything about a and b, I can say that every Haskell function has that type, because it has an input and it produces an output. I don't really know its usual name, but with it bottom (⊥) can be defined this way:

```
> f :: a -> b
> f x = f x
> bottom :: b
> bottom = f 0
```

We can use any value we want for bottom here.