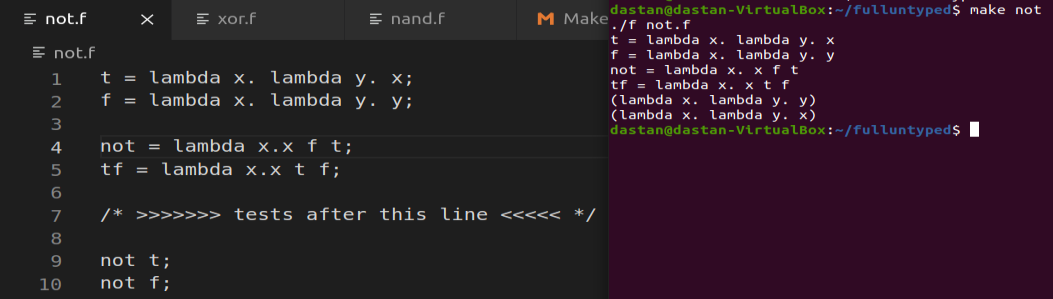
**Exercise 1.** Use Church Booleans to complete the functions:

***NOT = ƛx. x FT***

“If TRUE then TRUE else FALSE = TRUE”

“if FALSE the TRUE else FALSE = FALSE”

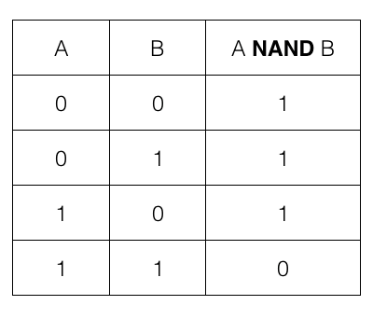
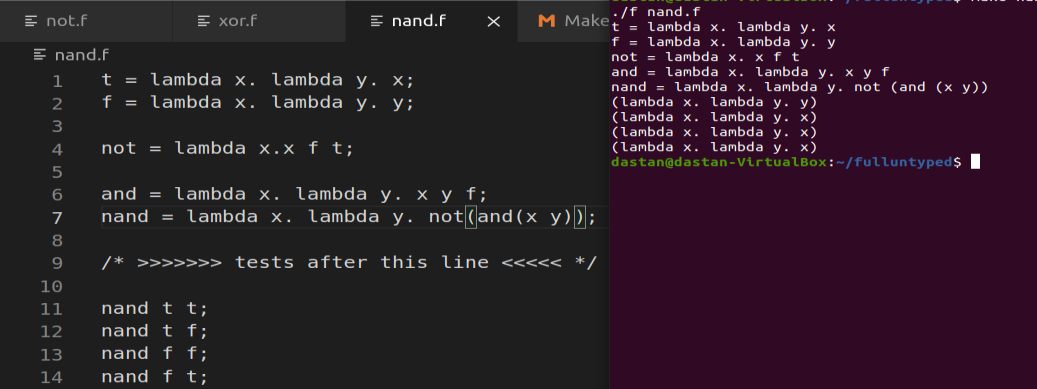
But our NOT function must do it vice versa. Therefor we just change the positions of TRUE and FALSЕ.



***NAND = ƛx. ƛy. x (NOT y) T***

For this function we must apply two variables, **T** OR **F**(Like (ƛx. ƛy. x (NOT y)T)TF).

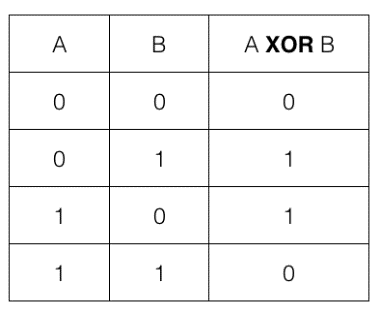
We know that NAND operation must give us next table which is vice versa for AND:

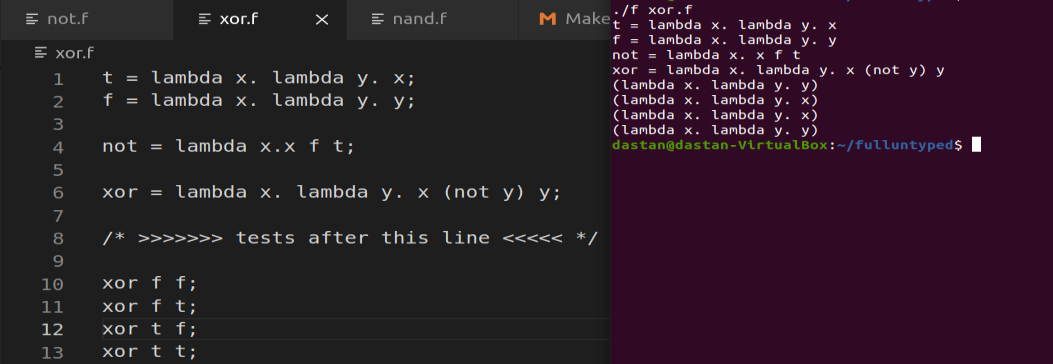


***XOR = ƛx. ƛy. x (NOT y) y***

For this function we must apply two variables, **T** OR **F**(Like (ƛx. ƛy. x(NOT y)y)TF).

We know that XOR operation must give us next table:



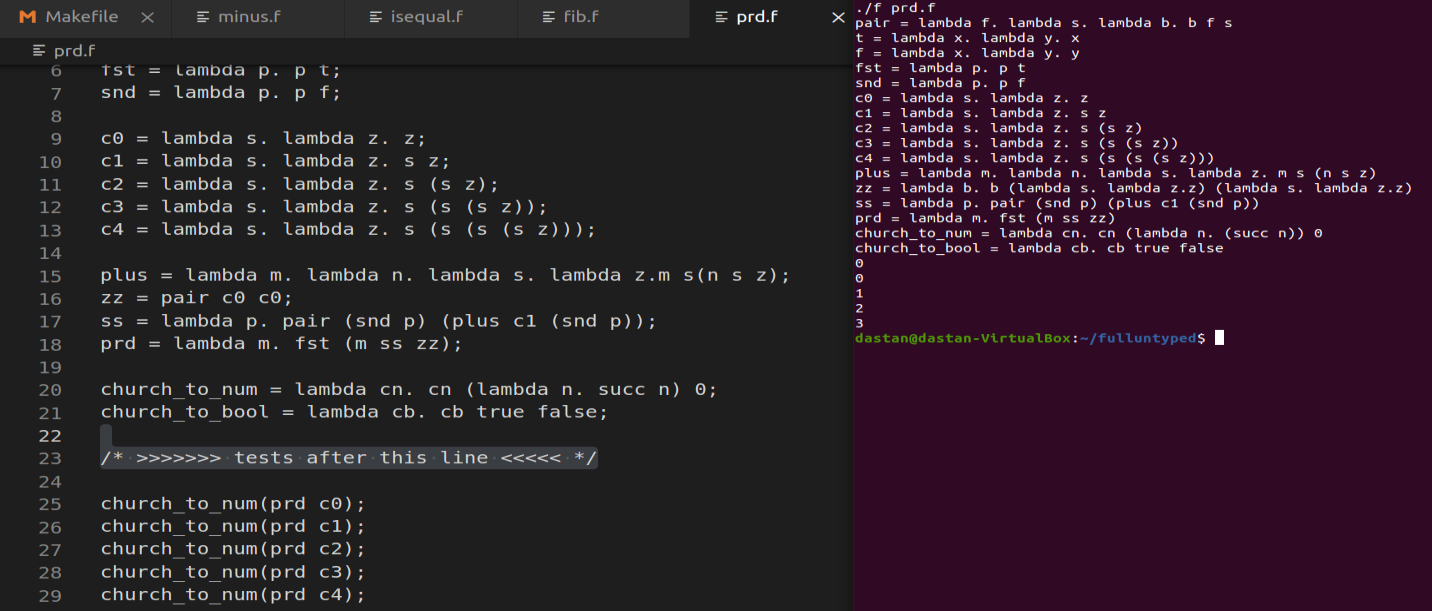


**Exercise 2.** using Churh Numerals:

***Pred =(ƛn. m fst (m ss zz)***

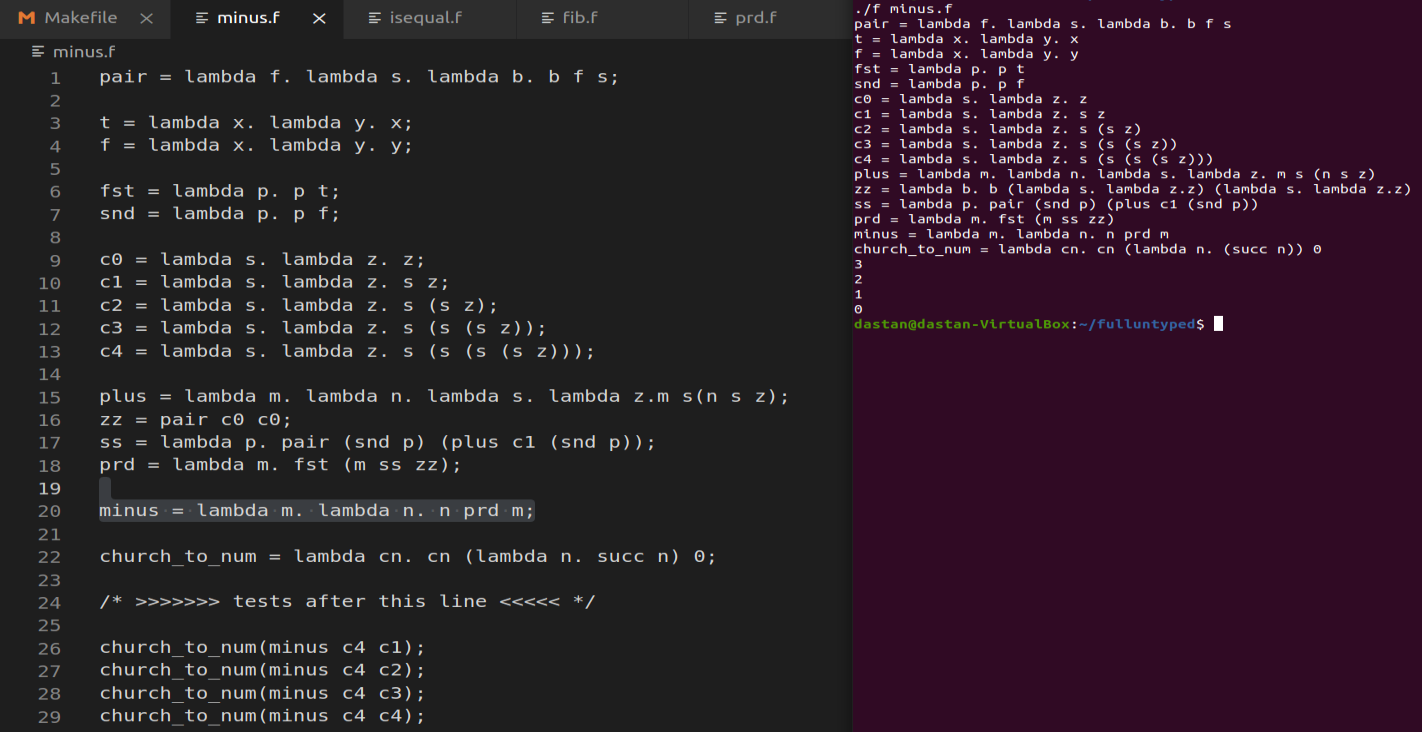
There we see operation which will show us next pair of numbers. Where first of the pair always will be smaller for one than the second number of pair. Because we just do a successor operation for the second number of the previous pair and past it as the second pairs second number. While, as a first number of new pair we will past the second number of the previous pair without successor operation.

Then, in our main predecessor function we see how we give a number n as a how many times this next pair operation must be completed.



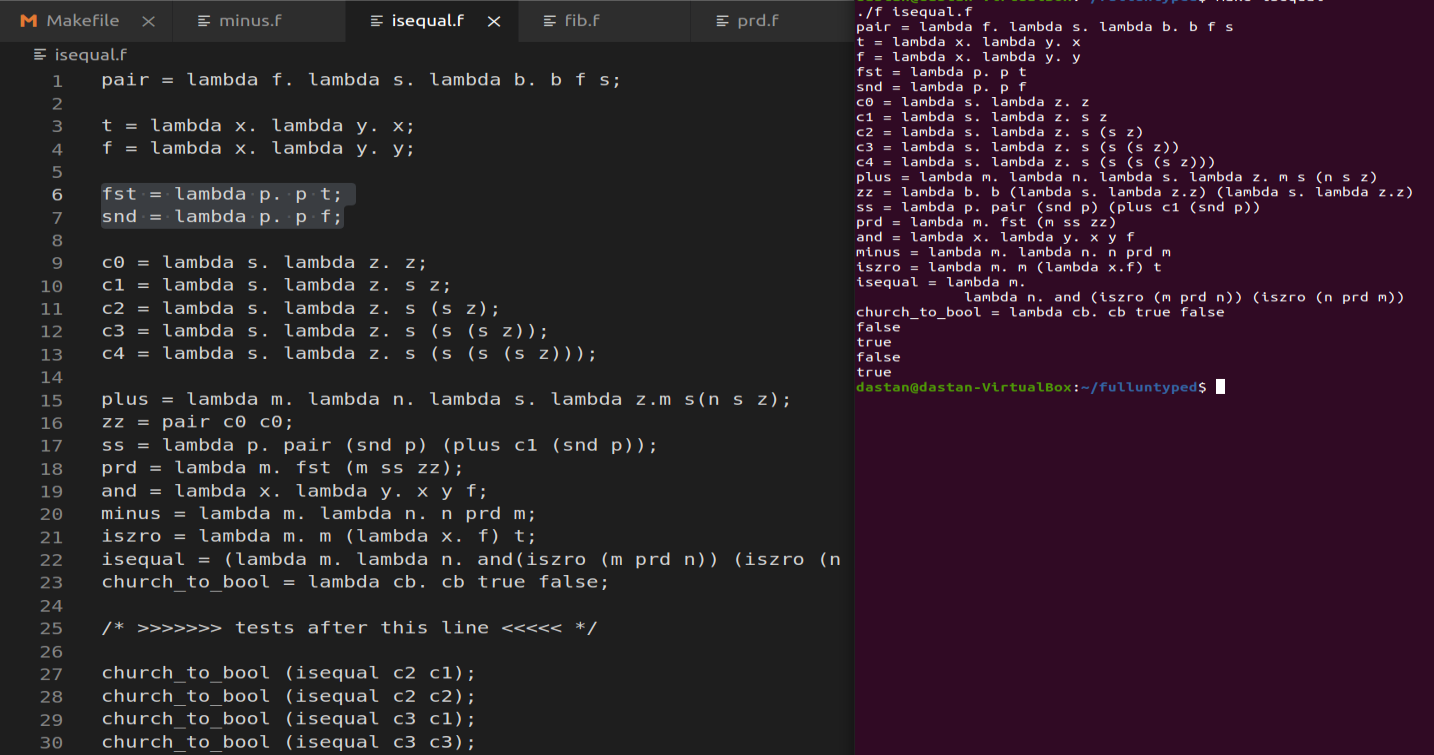
***Minus =(ƛn. ƛm. nm (n P m))***

This operation is very simple if predecessor operation is understanded well. We just apply n times predecessor function to m. If n will be greater, then answer will be zero. If n and m will be equal then answer also be zero. And only if n will be less than m, answer of subtraction function will be non-zero. Therefor it will be useful to also mention iszero function which will be done with a function which will apply a one number and will return T or F.



***Isequal =(ƛn. ƛm. AND(iszero(n P m) iszero(m P n)))***

This operation will use iszero operation which we mentioned in subtraction operation. Also, we see that we use AND operation which is ƛx. ƛy. xy F. By the knowledge from explanation of iszero, we know that for example, if we will have two numbers 4 and 5, and will apply it to iszero (4 P 5) and iszero (5 P 4). We will have FALSE for first and TRUE for second. Because 5 times done predecessor for 4 will give us 0. The reason is that there are no negative numbers applied. So only when we will have the same numbers n and m, we will have TRUE and TRUE.



**Exercise 3.** Implement function fib(n) that returns n-th Fibonacci number using

***a) Y-combinator*** => Y (f. ƛn. If (n <= 1) (ƛy. (f (n - 1)) + (f (n - 2))) (ƛy. n)) Where Y combinator is ƛf. (ƛx. f (x x)) (ƛx. f (x x)).

***b) Z-combinator*** => Z (f. ƛn. If (n <= 1) (ƛy. (f (n - 1)) + (f (n - 2))) (ƛy. n)) Where Z combinator is ƛf. (ƛx. f (ƛy. x x y)) (ƛx. f (ƛy. x x y)).

This two implementations might be seen that very same to each other. While the main point of difference is that Y combinator or in other words call-by-name fixed point combinator is completely useless in call by value setting. Because every time while we will compile Y combinator, it will not give us a value by default. For example: Y foo => foo (Y foo) => foo (foo (Y foo)) and so on. Therefor, there was implemented Z combinator which is also called call-by-value Y combinator. But I will call it like Z combinator. Real languages usually do call by value. So, Z combinator is useful when the first of all must be calculated the value of the given function. And only after value calculating, it will go on.

