**Лабораторна робота №3**

**Замикання і відкладені обчислення**

**Цілі роботи**: отримання навичок використання замикань та відкладених обчислень у мові F#.

**Завдання:**

Скласти функції для створення генераторів на основі застосування замикань.



Короткі теоретичні відомості

Карировані функції повертають функціональний тип за рахунок часткового застосування. Частина параметрів, які приймаються функцією, запам’ятаються у контексті функції і використовуються при подальших викликах.

Подібний механізм зв’язування функції, однак у відношенні до вільних імен, які не передаються через параметри, застосовується у всіх мовах функціонального програмування і називається лексичним замиканням (lexical closure), або просто замикання.

Подібне замикання є статичним, тому що у мові F# імена незмінні. У інших мовах функціонального програмування, наприклад у Lisp, зв’язування є динамічним і спільно із зміною значення імені здійснюється зміна результату застосування замикання. Така властивість замикання є корисною для створення генераторів: замикання використовується у якості закону отримання наступного значення, яке зберігається у контексті самого замикання.

У мові F# існує можливість оголошення змінних за допомогою ключового слова mutable. Однак для створення генераторі більш зручния є застосування спеціальних посилальних змінних типу ref. Для посилальних змінних передбачені три операції:

створення: х = ref <початкове значення>

витягування з ячейки !x

запис в ячейку: x := <нове значення>

Найпростіший генератор можна записати таким чином:

let gencount n=

let x = ref (n-1) in

fun() -> (x:=!x + 1; !x )

val gencount : int -> (unit -> int)

У цьому прикладі fun() лексичне замикання, контекст якого записується у дужках. Останній вираз – значення, яке повертає замикання при застосуванні. Одночасно значення змінної інкриментується.

Створення лічильника і застосування:

> let count = gencount 10;;

val count : (unit -> int)

> count();;

val it : int = 10

> count();;

val it : int = 12

**Рішення**

Аналіз задачі

Напишемо функцію *sinCosGen*, що буде отримувати початкове значення кута та крок та повертати генератор що повертає tuple зі значеннями синуса та косинуса. Напишемо два варіанти – через використання посилання (ref), та через mutable (починаючи з 4 версії в цих способах більше немає різниці, і Microsort рекомендує використовувати mutable, щоб F# код був більш консистентним <https://devblogs.microsoft.com/fsharpteam/announcing-a-preview-of-f-4-0-and-the-visual-f-tools-in-vs-2015/>).

Щоб вивід був нагляднішим згенеруємо за допомогою генератора список та форматуємо його.

Текс програми

let sinCosGen (initialAngle: float) step =

   let mutable angle = initialAngle - step;

   fun() -> (angle <- angle + step; (sin angle, cos angle))

let sinCosGenRef (initialAngle: float) step =

   let angle = ref (initialAngle - step);

   fun() -> (angle := !angle + step; (sin !angle, cos !angle))

let rec genlist n clo =

   if n = 0 then []

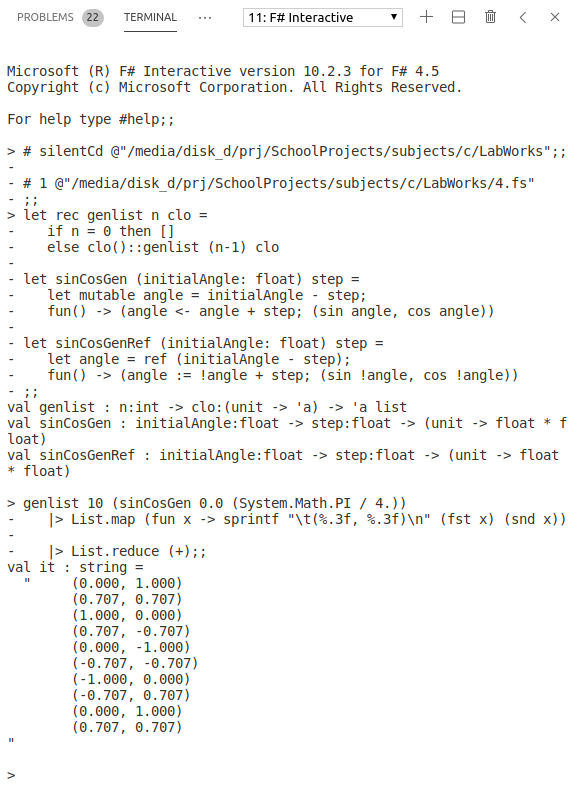
   else clo()::genlist (n-1) clo

genlist 10 (sinCosGen 0.0 (System.Math.PI / 4.))

   |> List.map (fun x -> sprintf "\t(%.3f, %.3f)\n" (fst x) (snd x))

   |> List.reduce (+)

Результат виконання програми



**Висновок:** на цій лабораторній роботі я отримав навички використання замикань та відкладених обчислень у мові F#.