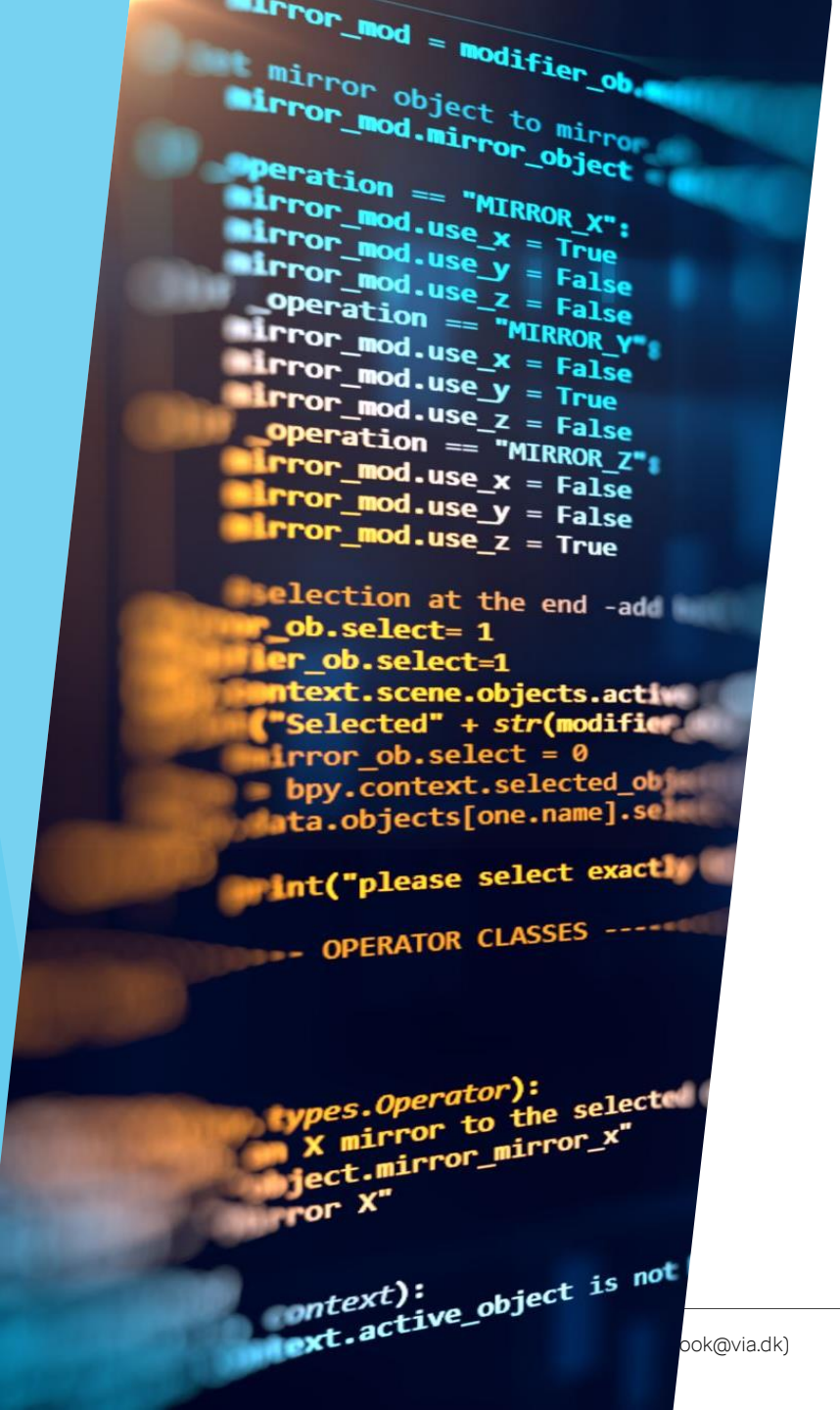


Programming Concepts and Languages

Spring 2024

Is this tail recursive?

```
let rec factorial x =
      if x <= 1
      then 1     // Base case
      else x * factorial (x - 1)
```

Is this tail recursive?

```
let rec sumList lst =
    match lst with
          | [] -> 0
          | hd::tl -> hd + sumList(tl)
```

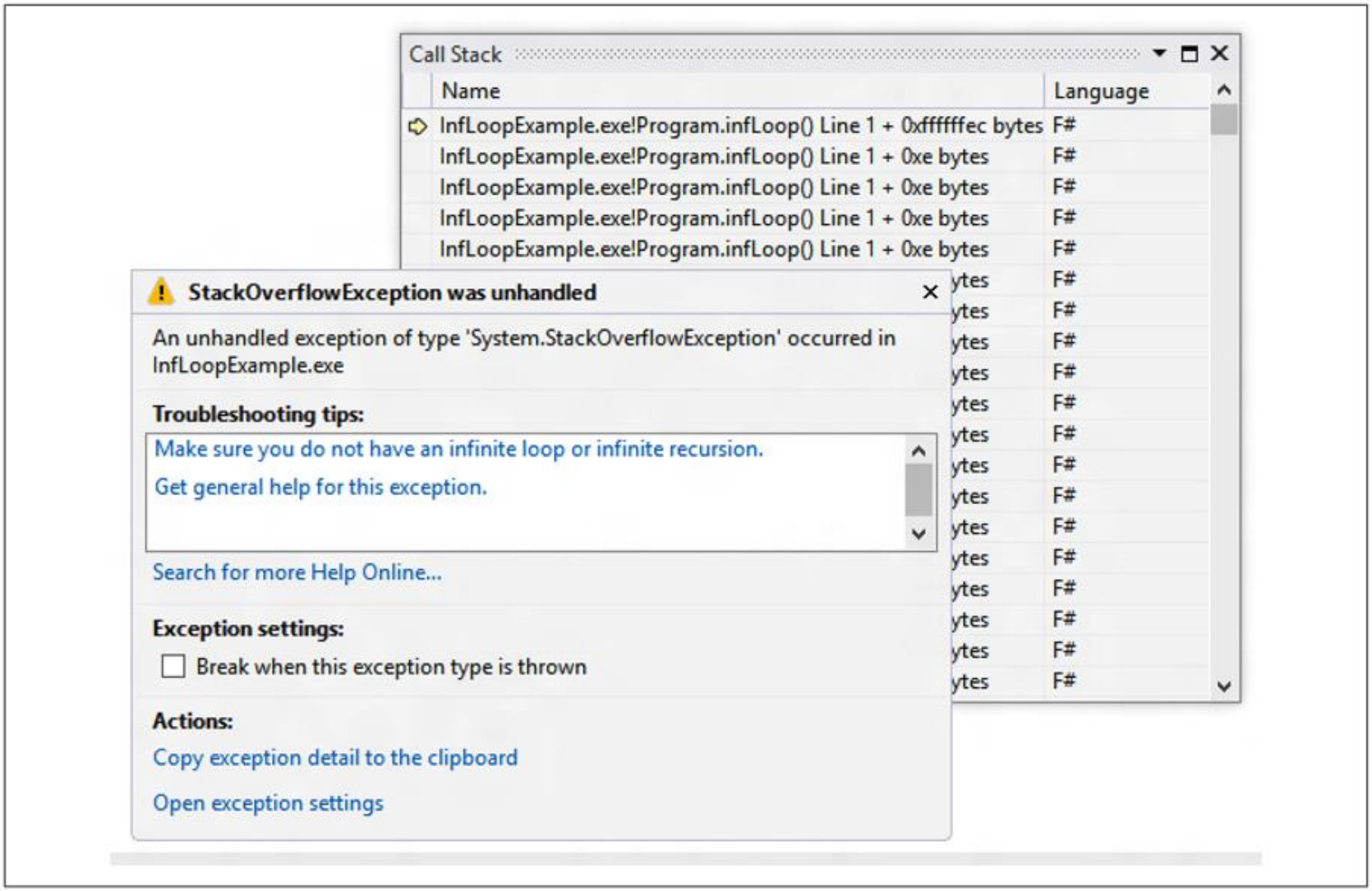
Is this tail recursive?

```
let rec fold_left f acc ls =
    match ls with
    | [] -> acc
    | l::ls' ->
          fold_left f (f acc l) ls'
```

Learning Objectives

- By the end of class today, you should be able to:
  - ✓ explain Tail Recursion
  - ✓ explain how to avoid stack overflows with tail recursion using accumulator and continuations
  - ✓ implement simple tail recursive F# programs
  - ✓ explain and implement simple F# programs using the following Recursive Data Types
    - ✓ Sequences
    - ✓ Sets
    - ✓ Maps
    - ✓ Arrays
- N/B Feedback to course project during exercises

Avoid Stack Overflow



Stack Frame

- For every function call, the runtime allocates a stack frame.
  - stored on a stack maintained by the system.
- A stack frame is removed when a call completes.
  - If a function calls another function, then a new frame is added on top of the stack.
- The size of the stack is limited, so too many nested function calls leave no space for another stack frame, and the next function can't be called.
- When this happens in .NET, a StackOverflowException is raised.

Is this tail recursive?

```
let rec sumList lst =
    match lst with
          | [] -> 0
          | hd::tl -> hd + sumList(tl)
```

Stack Overflow

The terminal process "/usr/bin/dotnet 'fsi'" terminated with exit code: 134.

Open Help

```
let rec sumList lst =  // list<int> -> int
    match lst with
        | [] -> 0
        | hd::tl -> hd + sumList(tl)

printfn "Sum: %d" (sumList [1..1000000])
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
Stack overflow.
Repeat 174401 times:
--------------------------------
   at FSI_0001.sumList(Microsoft.FSharp.Collections.FSharpList`1<Int32>)
--------------------------------
   at <StartupCode$FSI_0001>.$FSI_0001.main@()
   at System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Span`1<System.Object> ByRef, Sy
stem.Signature, Boolean, Boolean)
```

What can we do about it?

- The essential idea is that we only need to keep the stack frame because we need to do some work after the recursive call completes.

- Use **tail recursion**

```
let rec i_am_tail_recursive arg =
        (*check out of bound and fail with*)
        if(arg = 1000) then true
        else i_am_tail_recursive (arg + 1)
```

  - the last operation that ***i_am_tail_recursive*** function performs in the else branch is a recursive call.

- It doesn't need to do any processing with the result, it just returns it directly.
- This kind of recursive call is called tail recursion
- the result of the deepest level of recursion ***i_am_tail_recursive(1000)***, can be directly returned to the caller

Benefits of using Tail Recursion

- The function executes slightly faster, because fewer stack pushes and pops are required.
- The function can recurse indefinitely.
- No StackOverflowException is thrown.

- a function is considered tail recursive if and only if there is no work to be performed after a recursive call is executed

Tail-Recursive Patterns

Accumulator pattern

- add additional parameters
- pass an accumulator parameter to the recursive call so that the base case will return the final state of the accumulator.

Continuations – cont()

- rather than passing the current state of the accumulator "so far" as a parameter to the next function call, pass a function value representing the rest of the code to execute
- i.e., rather than storing "what's left" on the stack, you store it in a function.
- Continuations are function values that represent the rest of the code to execute when the current recursive call completes
- Conceptually, you are trading stack space (the recursive calls) with heap space (the function values).

Is this tail recursive?

```
let rec factorial x =
    if x <= 1
    then 1     // Base case
    else x * factorial (x - 1)
```

Analyzing the IL code

```
.method public static int32  factorial(int32 x) cil managed
{
  // Code size       17 (0x11)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.1
  IL_0002:  bgt.s      IL_0006
  IL_0004:  ldc.i4.1
  IL_0005:  ret
  IL_0006:  ldarg.0
  IL_0007:  ldarg.0
  IL_0008:  ldc.i4.1
  IL_0009:  sub
  IL_000a:  call       int32 session5::factorial(int32)
  IL_000f:  mul
  IL_0010:  ret
} // end of method session5::factorial
```

- mul instruction after the return from the recursive call to factorial, keeps us from utilizing tail call optimization.

Tail-recursive version

- By passing the data around as an extra parameter, you remove the need to execute code after the recursive function call
  - no additional stack space is required.
- When the F# compiler identifies a function as tail recursive
  - it generates the code differently.
- Example: the *tailRecursiveFactorial* function (next slide) would be generated as an iterative C# code, with a while loop.

Tail-recursive version - factorial

- Using accumulator

```
let accFactorial x =
    if x < 0 then failwith "Non natural number arg"
    // Keep track of both x and accumulator value (acc)
    let rec tailRecursiveFactorial x acc =
        if x <= 1 then
            acc
        else
            tailRecursiveFactorial (x-1) (acc * x)
    tailRecursiveFactorial x 1
```

Analyzing the IL code

```
.method public static int32  accFactorial(int32 x) cil managed
{
  // Code size       8 (0x8)
  .maxstack  8
  IL_0000:  ldarg.0
  IL_0001:  ldc.i4.1
  IL_0002:  call       int32 session5::tailRecursiveFactorial@19(int32,
                                                                 int32)
  IL_0007:  ret
} // end of method session5::accFactorial
```

- ✓ **`accFactorial`** code results in tail call optimization
- ✓ After a tail **call** instruction, no further code should execute other than **ret**.

Is this tail recursive?

```
let rec sumList lst =
    match lst with
        | [] -> 0
        | hd::tl -> hd + sumList(tl)
```

Redefine it to be tail recursive using Accumulator pattern

Avoiding Stack Overflow sumList

- Accumulator
  - When we use an accumulator argument, we need to write another function with an additional parameter

```
let sumListTR ls =
    let rec sumListHelper(ls, total) =
        match ls with
        | [] -> total
        | hd::tl ->
            let ntotal = hd + total
            sumListHelper(tl, ntotal)
    sumListHelper(ls, 0)
```

  - We don't usually want this to be visible to the caller, so we write it as a local function
  - The accumulator argument (*total*) stores the current result.
  - When we reach the end of the list, we already have the result, so we can just return it

Continuations

- Print a list of items in reverse using a continuation

```
let printListRev lst =
    let rec printListRevTR lst cont =
        match lst with
        // For an empty list, execute the continuation
        | [] -> cont()
        // for other lists, print the current node as part cont
        | hd :: tl ->
            printListRevTR tl (fun () -> printf "%A " hd
                                         cont())
    printListRevTR lst (fun () -> printfn "Done!")
```

```
printListRev ['A';'C';'T';'G']
```

- The printing is done as part of the continuation, which grows larger with each recursive call.
- In the base case, the continuation is called.

Recursive Data types

- Sequences
- Sets
- Maps
- Arrays
- FSharp.Collections includes:
  - Lists
  - Sequences
  - Immutable sets and maps based on binary trees
  - Arrays, including resizable ones (later –these are imperative).

Sequences

- Sequences are ordered-collections, like lists.
- However, unlike lists, sequences are evaluated as needed
- Sequences are defined just like list comprehensions.

```
let seqOfNumbers = seq { 1 .. 5 }
seqOfNumbers |> Seq.iter (printfn "%d")
```

- a list comprehension is just an abbreviation for a sequence comprehension followed by Seq.toList
- Sequences essentially allow generating a head element and a tail sequence, like the recursive function type:

```
type 'a seq = Seqof unit -> 'a * 'a seq
```

- The contents of lists are stored entirely in memory, whereas sequence elements are generated dynamically

Sequence of all integers

- define a sequence of all positive 32-bit integers

```
let allPositiveIntsSeq =
   seq { for i in 1 .. System.Int32.MaxValue do yield i }
```

```
seq<int> = seq [1; 2; 3; 4; ...]
```

- defining an equivalent list

```
let allPositiveIntsList =
                [ for i in 1 .. System.Int32.MaxValue -> i ]
```

- fails due to memory

```
ERROR!!!
System.OutOfMemoryException: Exception of type
System.OutOfMemoryException' was thrown.
```

Sequence Expressions

- We can use the same list comprehension syntax to define sequences (technically referred to as sequence expressions

```
seq { for i in 1 .. 100 -> (i,i*i) }
// same as above
Seq.map (fun i -> (i,i*i)) { 1 .. 100 }
```

- Sequence Module Functions
  - Seq.take: Returns the first n items from a sequence
  - Seq.unfold: Seq.unfold generates a sequence using the provided function
  - Seq.iter: Iterates through each item in the sequence
  - Seq.map Produces a new sequence by mapping a function onto an existing sequence
  - Seq.fold Reduces the sequence to a single value
    - Because there is only one way to iterate through sequences, there is no equivalent to List.foldBack:

Some functions on Sequences

- F# has a module Seq with functions on sequences. Many of these have
- counterparts for lists and arrays.
  - `Seq.length : seq<'a> -> int`
  - `Seq.append : seq<'a> -> seq<'a> -> seq<'a>`
  - `Seq.take : int -> seq<'a> -> seq<'a>`
  - `Seq.skip : int -> seq<'a> -> seq<'a>`
  - `Seq.zip : seq<'a> -> seq<'b> -> seq<'a * 'b>`
  - `Seq.filter : ('a -> bool) -> seq<'a> -> seq<'a>`
  - `Seq.map : ('a -> 'b) -> seq<'a> -> seq<'b>`
  - `Seq.fold : ('a -> 'b -> 'a) -> 'a -> seq<'b> -> 'a`
- Examples:

```
Seq.map (fun i -> (i,i*i)) { 1 .. 100 }
Seq.fold (+) 0 { 1 .. 100 }
```

Seq Module Functions

Function and Type	Description
`Seq.length` `seq<'a> -> int`	Returns the length of the sequence.
`Seq.exists` `('a -> bool) -> seq<'a> -> bool`	Returns whether or not an element in the sequence satisfies the search function.
`Seq.tryFind` `('a -> bool) -> seq<'a> -> 'a option`	Returns `Some(x)` for the first element `x` in which the given function returns `true`. Otherwise returns `None`.
`Seq.filter` `('a -> bool) -> seq<'a> -> seq<'a>`	Filters out all sequence elements for which the provided function does not evaluate to `true`.
`Seq.concat` `(seq< #seq<'a> > -> seq<'a>`	Flattens a series of sequences so that all of their elements are returned in a single `seq`.

List, Arrays and Sequences

- `'a` list and `'a[]` are *subtypes* to `seq<'a>`
  - i.e. functions taking sequences as arguments can be given lists or arrays as arguments instead
- Examples:

```
Seq.map (fun x -> x +1) [19;29;39] gives [20;30;40]

Seq.zip [|1; 3; 5|] [| 'a' ;'b' ;'c'|]
gives [(1, 'a'); (3, 'b'); (5, 'c')]
```

- **Defining Lists and Arrays by Sequence Expressions**
  - Sequence expressions can be used to define lists or arrays

Simply write "`[ ... ]`" or "`[| ... |]`" rather than "`seq { ... }`"

```
[1 .. 5] gives  [1; 2; 3; 4; 5]

[|1 .. 5|] gives [|1; 2; 3; 4; 5|]
```

- converting an array to a list:

```
let array2list a = [for i in 0 .. Array.length a - 1
-> a.[i]]
```

Sets

- Sets are a useful alternative to lists in some situations.
- They are implemented with binary trees, so allow fast checking for membership, and avoid duplicates.
- They can be created by starting from an empty list, and adding elements.
- Fsharp.Collections.Set has a couple of functions:
  - Union, difference, ...
  - Member, subset
  - Conversion to and from lists (in sorted order)
- Example:

```
let a = Set.ofSeq [ 1 .. 10 ]
let b = Set.ofSeq [ 5 .. 10 ]
Set.isSubset b a  gives true
```

Maps

- similar to sets, except for the “key” and an associated “value”.
- Example:

```
let comingEvents =
    [ ("Palmesøndag", "March 24");
      ("Skærtorsdsag", "March 28");
      ("Langfredag", "March 29");
      ("Påskedag", "March 31")
    ]
    |> Map.ofList
```

- How do I get the date for “Langfredag” (Good Friday)?

```
comingEvents.["Langfredag"] = "March 29”
```

- See FSharp.Collections.Map for details of the functions available, including *add*, *exists*, *filter*, *find*, *remove*, *tryfind*.