

# Programmering og Problemløsning

## Datalogisk Institut, Københavns Universitet

### Arbejdsseddel 7 - individuel opgave

Fritz Henglein, Mads Obitsø

18. november - 26. november.  
Afleveringsfrist: lørdag d. 26. november kl. 22:00.

I denne periode skal vi arbejde med funktionelle datastrukturer (eng. *purely functional data structures*), inddata/uddata (eng. *input/output, IO*) og undtagelser (eng. *exceptions*).

Denne arbejdsseddels læringsmål er:

- Implementering af funktionelle datastrukturer ved hjælp af induktive datatyper
- Skrive programmer der kan håndtere input og genere output, ved at læse fra og skrive til filer.
- Skrive programmer der kan håndtere undtagelser, f.eks. ifbm. med at læse og skrive til filer.

Opgaverne er opdelt i øve- og afleveringsopgaver. I denne periode skal I arbejde individuelt med jeres afleveringsopgaver. Regler for gruppe- og individuelle afleveringsopgaver er beskrevet i “Noter, links, software m.m.” → “Generel information om opgaver”.

## Øveopgaver (in English)

### Difference lists and equational rewriting

A strong point of *purely* functional programming, that is programming with functions that have no side-effects, is that they satisfy algebraic equalities that can be applied to rewrite expressions and yet be guaranteed to be observationally equivalent even when the expression contains variables whose binding is unknown.<sup>1</sup>

7ø0 Rewriting can help correctly derive more efficient code or just code that may be more easily understandable in terms of more elementary operations.

Consider the compact linear-time implementation for computing the preorder of a binary tree

---

<sup>1</sup>This is a hallmark of mathematical notation and functional programming.

```

type 'a tree =
  Empty | Leaf of 'a | Branch of 'a tree * 'a * 'a tree
open DiffList // nil, single, append, fromDiffList
let combinePre (dl1, x, dl2) = append (single x) (append dl1 dl2)
let preorder' t = treeFold (nil, single, combinePre) t
let preorder t = fromDiffList (preorder' t)

```

where `treeFold` from the lecture slides of lecture week 8 is defined as

```

let treeFold (e, lf, bf) =
  let rec f t =
    match t with
    | Empty -> e
    | Leaf x -> lf x
    | Branch (t1, x, t2) -> bf (f t1, x, f t2)
  f

```

See `DiffList.fs` in the F# code distributed in lecture week 8 for the definitions of difference list operations `nil`, `single`, `append` and `fromDiffList`.

Rewrite `preorder'` starting with the expression

```
preorder' t xs
```

by *unfolding* and *folding* definitions of functions to arrive at a definition of `preorder'` of the form

```

let preorder' t xs =
  match t with
  | ... -> ...
  | ... -> ...
  | ... -> ... preorder' ... preorder'

```

In particular, unfold all definitions of functions on difference lists.

Unfolding is the process of replacing a term of the form `f ...` with the corresponding right-hand side of the definition of function `f`. Folding is the converse.

Finally, do the same for `preorder` by rewriting

```
preorder t xs
```

## Catenable lists

Catenable lists are lists with efficient (constant-time) appending, like difference lists, and additional operations. They are widely used to implement text processing systems such as text editors, where characters and text fragments need to be inserted and deleted efficiently, which is why arrays holding the text are not used.

In this exercise you will implement a module `CatList` with functional catenable lists, using inductive data types in F#.<sup>2</sup>

Start by creating a new dotnet project using

```
dotnet new console -lang "F#" -o CatenableLists,
```

then create the files `CatList.fsi` and `CatList.fs` and add them to `CatenableLists.fsproj`.

## 7ø1 Inductive data type and constructors

We represent catenable lists by the inductive data type

```
type 'a catlist =  
    Empty                                // empty node  
    | Single of 'a                      // leaf node  
    | Append of 'a catlist * 'a catlist // internal node
```

The constructor `Empty` represents the empty list; `Single` constructs a singleton list; `Append` constructs the concatenation of two lists.

Provide definitions (“one-liners”) for the following values and functions.

```
val nil : 'a catlist // the empty list  
val single : 'a -> 'a catlist // singleton list  
val append : 'a catlist -> 'a catlist -> 'a catlist // append  
val cons : 'a -> 'a catlist -> 'a catlist // cons/prepend  
val snoc : 'a catlist -> 'a -> 'a catlist // snoc/postpend
```

Use these functions instead of `Empty`, `Single`, `Append` in subsequent code, except in pattern matching.

## 7ø2 Tree traversal by structural recursion

The length of a catenable list can be defined by structural recursion on `'a catlist`:

```
let rec length' xs =  
    match xs with  
    | Empty -> 0  
    | Single _ -> 1  
    | Append (ys, zs) -> length' ys + length' zs
```

Define in an analogous fashion the function `sum' : int catlist -> int`, which computes the sum of the integer values in its input. Test that it computes the correct results on carefully chosen inputs, including the “extremal” value `nil`.

---

<sup>2</sup>We use the term “list” in the programming language independent sense of “finite sequence of elements”. If we want to refer to the built-in F# data type `someType list` we say “built-in cons-lists in F#”, but may elide “built-in” and “in F#” where this is clear from the context.

### 7ø3 Tree traversal by folding

Structural recursion on 'a catlist can be captured by a parameterized higher-order function

```
fold: (('a -> 'a -> 'a) * 'a) -> ('b -> 'a) -> 'b catlist -> 'a
```

such that the length function can be defined by

```
let length xs = fold ((+), 0) (fun _ -> 1) xs
```

without using “rec” in its definition.

Define fold by structural recursion analogous to treeFold for binary trees (see lecture slides).

### 7ø4 Tree folding examples

Analogous to the fold-based definition of length above, define using fold, without explicit recursion, the functions on catenable lists that correspond to the functions of the same names on built-in cons-lists.<sup>3</sup>

```
val map : ('a -> 'b) -> 'a catlist -> 'b catlist
val filter : ('a -> bool) -> 'a catlist -> 'a catlist
val rev : 'a catlist -> 'a catlist
```

---

## File concatenation

The cat-utility from Unix<sup>4</sup> is a program that concatenates files. This exercise is about building a cat-like program in F# in a file called cat.fs that contains the following functions, plus additional definitions as you see fit to solve this exercise.

Start by creating a new dotnet project using

```
dotnet new console -lang "F#" -o cat,
```

then create the files Cat.fsi and Cat.fs and add them to cat.fsproj.

You can then use the following *skeletons* for the three files:

- **Cat.fsi**

---

<sup>3</sup>Tip: Write the functions using structural recursion first; then identify the parts that become the arguments of fold. Finally write the functions using fold and test them against your first version to ensure they give the same result.

<sup>4</sup>Unix is the predecessor operating system for MacOS, Linux and most server operating systems in current use.

```

module Cat
open System.IO

val readBytes: FileStream -> byte[]
val readFile: string -> byte[]
val readFiles: string list -> byte[] option list
val writeBytes: byte[] -> FileStream -> unit
val writeFile: byte[] -> string -> int
val cat: string[] -> int

```

- **Cat.fs**

```

module Cat
open System.IO

let readBytes (fs:FileStream) : byte[] =
    [||] // Replace this with a proper implementation

let readFile (filename:string) : byte[] =
    [||] // Replace this with a proper implementation

let readFiles (filenames : string list) : byte[] option list =
    [] // Replace this with a proper implementation

let writeBytes (bytes : byte[]) (fs:FileStream) =
    () // Replace this with a proper implementation

let writeFile (bytes: byte[]) (filename:string) =
    0 // Replace this with a proper implementation

let cat (filenames : string[]) =
    0 // Replace this with a proper implementation

```

- **Program.fsx**

```

open Cat

[<EntryPoint>]
let main (args : string[]) =
    // args is a string array
    // containing the command-line arguments
    printfn "%A" args
    0 // The exit code, 0 means "all is good"

```

## 7ø5 Reading the contents of a file stream

Write function `readBytes: FileStream -> byte[]` with the following specification.

- Precondition: `fs` is a readable file stream.
- Postcondition: For `bs = readBytes fs`, the byte array `bs` contains the entire contents of the file stream.

You should create a byte array of a suitable size and use `FileStream.Read((buffer: byte[]), (offset: int32), (count: int32))` to read the file contents into the array before returning it.

## 7ø6 Reading contents of file

Implement a function `readFile: string -> byte[]` with the following specification.

- Precondition: None (any string is acceptable and must be handled)
- Postcondition: If the input string is a readable file, return its contents. Otherwise raise exception `FileNotFoundException` if the file does not exist or is not readable.

The function should obtain a `FileStream`, e.g. through `File.OpenRead`, and use your `readBytes` to read the filestream.

Note: While it is possible to implement this function using `File.ReadAllBytes`, the intended learning goal of this exercise is how to *implement your own version*.

## 7ø7 Reading contents of multiple files

Implement the function `readFiles: string list -> byte[] option list` with the following specification:

- Precondition: None (any list of strings is acceptable and must be handled)
- Postcondition: For each string in the input, the output is either the contents of the file with that name (wrapped in `Some`), if the file exists and is readable; otherwise it is `None`.

## Writing bytes to a file stream

Write function `writeBytes: byte[] FileStream -> unit` with the following specification.

- 7ø8
- Precondition: The file stream is readable.
  - Postcondition: all bytes in `bytes` are written to `fs`.

## 7ø9 Writing bytes to file

Implement a function `writeFile: byte[] string -> int` with the following specification.

- Precondition: None.

- Postcondition: All bytes in the byte array are written to the file with the given name. If the file does not exist, it is created. If it exists, it is overwritten with the contents of the byte array and the exit status is 0. If an error occurs, the string  
`cat: Could not open or create file filename .\n` is written to stderr. The exit status is 1.

The function should obtain a `FileStream` and use `writeBytes` to write the bytes to the filestream. You should use `File.Open` with an appropriate `FileMode`. You might need to call the `Flush()` method on your `FileStream` after writing bytes to it.

## 7ø10 Concatenating file contents and writing to files

Implement a function `cat: string[] -> int` that outputs to the last file name in the argument array the concatenation of the contents of all files in the argument array except for the last one, in the sequence they occur. Its specification is as follows.

- Precondition: None (any array of strings is acceptable and must be handled). Note that the same string (file name) can occur multiple times in input string array.
- Postcondition:
  - If all of the input files exist and are readable, the output written to the last file name contains their concatenated contents in the order given in the input array. Nothing is written to stderr and the exit status (result of the function) is 0.
  - If one or more of the input files does not exist or is not readable, then nothing is written to the last filename. The exit status is  $k$  where  $k$  is the minimum of 254 and the number of nonexistent, unreadable or unwritable files. For each string  $s$  that is a nonexistent, unreadable or unwritable file, the string  
`cat: The file s does not exist, is not readable or is not writeable.\n` is written to stderr.
  - If the string array contains only an output file name, that is has length 1, the output file must contain the empty string if it is writable, and the exit status is 0. (The empty string is the neutral element for strings under concatenation.)
  - If the string array is empty, the string `cat: no output file\n` is written to stderr, and the exit status is 255.

## Putting it all together

In `Program.fs` call `cat` with the command line arguments.

`dotnet run file1.txt file2.txt file3.txt` should result in `file3.txt` being either created or overwritten, and should contain the concatenated contents of `file1.txt` and `file2.txt`.

Test your implementation using specification-based testing, such that each postcondition is covered by at least one test case, including relevant extremal values; f.eks. ensuring that your implementation returns exit code 254 when concatenating 255 or more input files.

# Afleveringsopgaver (in English)

## Catenable list operations

These exercises continue with catenable lists.

### 7i0 Conversion to and from cons-lists

Write functions

```
val fromCatList : 'a catlist -> 'a list
val toCatList : 'a list -> 'a catlist
```

for converting between built-in cons-lists and catenable lists in linear time, first using structural recursion on catenable lists and built-in lists, respectively.<sup>5</sup>

Then express your definitions without explicit recursion, using the higher-order functions `List.foldBack` and `CatList.fold` instead.

(Optional: Your function `toCatList` will probably construct skewed binary trees. The function `balTree` presented in the lecture constructs a balanced search tree that. By analogy construct a balanced catenable list in time  $\Theta(n \log n)$ .

(Optional challenge: Come up with an implementation for `toCatList` that yields a balanced catenable list in linear time, that is in time  $O(n)$ .)

### 7i1 Looking up, inserting and deleting elements

Provide implementations, using explicit recursion, of functions

```
val item : int -> 'a catlist -> 'a
val insert : int -> 'a -> 'a catlist -> 'a catlist
val delete : int -> 'a catlist -> 'a catlist
```

where `item i xs` returns the  $i+1$ -th element in `xs` under the assumption (precondition) that  $0 \leq i < \text{length } xs$ ; `insert i v xs` inserts `v` after the  $i$ -th element in `xs`, under the assumption that  $0 \leq i \leq \text{length } xs$ ; and `delete i xs` deletes the  $i+1$ -th element in `xs` under the assumption that  $0 \leq i < \text{length } xs$ .

You may use the function `length : 'a catlist -> int` in your definitions. This makes your implementation slow, but is okay since it can subsequently be implemented in constant time by data augmentation. Length-augmented catenable lists or sometimes called *ropes*.

Using an inefficient, but correct implementation, as in this exercise, is a valuable intermediate step in the systematic design of efficient data structures.

---

<sup>5</sup>Tip: Use difference lists as an intermediate data structure when converting from catenable lists.



## Streaming file concatenation

Reading the contents of all files into memory before writing to the output stream requires memory proportional to the collective size of all files. Imagine we want to concatenate 10 files, each of size 4GB, using the `cat`-implementation from the exercises. We would need at least 40GB of memory to read in the files, before we could start concatenating the contents and writing the concatenation a file.

In this task you will write another implementation of the `cat`-program from the exercises, this time using *streaming*<sup>6</sup>.

- 7i2 Provide another implementation of `cat: string[] -> int` that uses only a constant amount of memory, 64 bytes as a buffer for data read. Note that you must satisfy the same specification for `cat`; in particular, nothing is to be written to `stdout`. Errors should be written to `stderr`.

Use the code in the `StreamingCat` directory in `7i_handout.zip` as a starting point.

You *must* implement the functions `cat` and `catWithBufferSize`.

## Krav til afleveringen

Afleveringen skal bestå af

- en zip-fil, der hedder `7i.zip`

Zip-filen skal indeholde:

- en mappe `CatList` med følgende filer:

`CatList.fsproj`, `CatList.fs`

Filen `CatList.fs` skal indeholde type definitionen

```
type 'a catlist = Empty | Single of 'a | Append of 'a catlist
* 'a catlist
```

samt funktionsdefinitionerne for de 5 påkrævede funktioner

```
val fromCatList : 'a catlist -> 'a list
val toCatList : 'a list -> 'a catlist
val item : int -> 'a catlist -> 'a
val insert : int -> 'a -> 'a catlist -> 'a catlist
val delete : int -> 'a catlist -> 'a catlist
```

og andre funktioner brugt for at implementere dem;

- en mappe `CatStreaming` med følgende filer:

---

<sup>6</sup>Somewhat similar to how a *streaming service* such as Youtube or Netflix delivers small pieces of a video in a *stream*, so the user does not have to wait for the entire video to be delivered, but can start watching (almost) instantly.

CatStreaming.fsproj, CatStreaming.fsi, CatStreaming.fs, Program.fs, testing.txt

Udover filerne skal mappen CatStreaming indeholde en mappe testFiles, der indeholder de input og output-filer du har benyttet til at teste din implementation. Filen testing.txt skal beskrive hvordan du afprøvet din løsning, herunder hvilke bufferstørrelser du benyttet, og beskrivelser af dine testfiler. Projektet skal kunne bygges med dotnet build og køres med dotnet run [fil1 fil2 ... filN].

Funktionerne skal være dokumenteret ifølge dokumentationsstandarden ved brug af <summary>, <param> og <returns> XML tagsne, i filen CatStreaming.fsi.

God fornøjelse.