Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Arbejdsseddel 7 - individuel opgave

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

I denne periode skal vi arbejde med ...

Denne arbejdsseddels læringsmål er:

- Input/Output
- Undtagelser

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.[1]

[1] This is a hallmark of both mathematical notation and functional programming. No “real” programming language with (implicit) side effects, however, has this property; thus secure and correct *automatic* program analysis, program synthesis and optimization is vastly more difficult with such programming languages. Indeed, while the level of abstraction has been lifted substantially over the last fifty or so years, bringing more automation and security than previously to bear, programming is still largely a manual process of humans hand-typing code into an editor — which is a somewhat paradoxical aspect: Even with modern of even modern software development methods (which are technology supported, but still largely manual) the very discipline that automates engineering and processing in *other* disciplines is itself still largely manual craftsmanship.

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

```
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 : 'a tree) : 'a difflist =
    treeFold (nil, single, combinePre) t
let preorder (t : 'a tree) : 'a list =
    fromDiffList (preorder' t)
```

using difference list.

Rewrite `preorder'` starting with the definition

```
let preorder' (t : 'a tree) (xs : 'a list) : 'a list =
    match t with
      Empty ->
         treeFold (nil, single, combinePre) Empty xs
    | Leaf x ->
         treeFold (nil, single, combinePre) (Leaf x) xs
    | Branch (t1, x, t2) ->
         treeFold (nil, single, combinePre) (Branch (t1, x, t2))
   xs
```

by first *unfolding* the occurrences of `treeFold`, that is replacing it with its definition; then inlining `nil`, `single`, `combinePre`, `append` until all these occurrences are eliminated; and finally *folding* occurrences of `treeFold (nil, single, combinePre)`, that is replacing them by `preorder'`.

Finally, do the same for `preorder` by inlining `fromDiffList`, thus eliminating all occurrences of difference list operations. (See DiffList.fs for the definitions of difference list operations.)

7ø1 File concatenation

The `cat`-utility from Unix[2] 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.

We recommend that you create 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:

[2] Unix is the predecessor operating system for MacOS, Linux and most server operating system in current practical use.

- **Cat.fsi**

```
module Cat
```

- **Cat.fs**

```
module Cat
```

- **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"
```

Reading contents of file stream

Write function `readBytes: fs: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.

Reading contents of file

Implement a function `readFile: filename: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` and use `readBytes` to read the filestream.

Reading contents of multiple files

Implement a function `readFiles: (filenames: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, if it exists and is readable; otherwise it is `None`.

Writing bytes to a file stream

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

- Precondition: `bytes` is a byte array and `fs` is a readable file stream.
- Postcondition: all bytes in `bytes` are written to `fs`.

Writing bytes to file

Implement a function `writeFile: (bytes:byte[]) (filename:string) -> ()` with the following specification.

- Precondition: `bytes` is a byte array.
- Postcondition: All bytes in `bytes` are written to the file `filename`. If `filename` does not exist, it is created. If it does exist, it is overwritten with the contents of `bytes`.

The function should obtain a `FileStream` and use `writeBytes` to write the bytes to the filestream.

Concatenating file contents and writing to files

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

- Precondition: None (any array of strings is acceptable and must be handled).
- Postcondition:
  - If all of the input files exist and are readable, the output written to the last `filename` 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 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 255 and the number of nonexistent/unreadable files. For each string *s* that is a nonexistent/unreadable file, the string
    `cat: The file` *s* `does not exist or is not readable.\n`
    is written to stderr.
  - If `filenames` contain a single element, that file is either created or overwritten with nothing. The concatenation of "nothing" is the empty string.
  - If `filenames` is empty, the string `cat: no input files\n` is written to stderr. The exit status is 0.

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`.

Specification-based testing of functions

Add test cases to the test cases given such that each part of the specifications is covered and such that extremal values of inputs are considered, such as empty files (files that exist and are readable, but contain zero bytes). In particular, test your implementation of `cat`.

7ø2 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#.[3]

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 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.

[3] We use the term "list" in a 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.

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 is computes the correct result on a carefully chosen inputs, including the "extremal" value `Empty`.

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).

Tree folding examples

Analogous to the `fold`-based definition of `length` above, define, without explicit recursion, the functions on catenable lists that correspond to the functions of the same names on built-in cons-lists, using `fold`.[4]

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

Afleveringsopgaver (in English)

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.

[4]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.

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 if there is a nonexistent/unreadable file in the input.

7i0 Reading the contents of all files into memory before writing to the output stream requires memory proportional to the collective size of all files. 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 if there is a nonexistent/unreadable file in the input.

7i1 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, using structural recursion on catenable lists and built-in lists, respectively.[5]

(Optional: Express your definitions without explicit recursion, using the `List.foldBack` and `CatList.fold` higher-order functions instead.)

(Optional challenge problem: The function `toCatList` will construct skewed binary trees. The function `balTree` presented in the lecture constructs a balanced search tree that, by analogy, yields a way of constructing a balanced catenable list, but in time $\Theta(n \log n)$. Come up with an implementation for `toCatList` that yields a balanced catenable list in time $O(n)$.)

7i2 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 <= i < length xs`; `insert i v xs` inserts `v` after the `i`-the element in `xs`, under the assumption that `0 <= i <= length xs`; and `delete i xs` deletes the `i+1`-th element in `xs` under the assumption that `0 <= i < 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. (See following optional exercise.) Using such an inefficient implementation is a valuable intermediate step in the systematic design of efficient data structures.

[5] Tip: Use difference lists as an intermediate data structure when converting from catenable lists.

Krav til afleveringen

Afleveringen skal bestå af

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

Zip-filen skal indeholde:

- filen `README.txt` som er en textfil med jeres navne og dato arbejdet, samt en beskrivelse af, hvordan man kører jeres kode.
- en `src` mappe med følgende og kun følgende filer: Funktionerne skal være dokumenteret ifølge dokumentationsstandarden ved brug af `<summary>`, `<param>` og `<returns>` XML tagsne.

God fornøjelse.