Programmering og Problemløsning
Datalogisk Institut, Københavns Universitet
Arbejdsseddel 5 - individuel opgave

7. oktober - 15. oktober 2022.
Afleveringsfrist: lørdag d. 15. oktober, 2022, kl. 22:00.

I denne periode skal vi arbejde med abstrakte datatyper. En *abstrakt datatype* er en type, der specificeres ved at angive operationer på typen og de egenskaber disse operationer opfylder. Med andre ord, en abstrakt datatype siger, hvordan den kan bruges af en programmør, uden at vedkommende behøver at vide, hvordan den er implementeret. En implementering af en abstrakt datatype er en konkret type og konkrete funktioner (funktionsdefinitioner som kode), der skal opfylde de givne egenskaber.

I F# implementeres abstrakte datatyper ved hjælp af *moduler*. Et modul består af en signaturfil med definitionen af typen og navnene af operationerne og deres typer; en implementationsfil, der indeholder operationernes definition, dvs. deres kode; og en applikationsfil, som typisk indeholder anvendelser af operationerne for at afprøve, hvorvidt implementeringen af operationerne overholder de krævede egenskaber. Moduler kaldes også *biblioteker*, fordi de samler nogle operationer og stiller dem til rådighed for brugere (programmører), lige som et bibliotek samler bøger og stiller dem til rådighed for brugere (læsere).

Denne arbejdsseddels læringsmål er:

- at lave et bibliotek og en tilhørende applikation,
- at kunne oversætte biblioteksfiler og applikationsfiler både med `dotnet fsi` og `run`,
- at kunne arbejde med generiske moduler,

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)

5ø0 In an earlier assignment, you implemented a small set of functions for vector operations in F#:

(a) addition of vectors

```
add: vec -> vec -> vec
```

(b) multiplication of a vector and a floating-point number

```
mul: vec -> float -> vec
```

(c) rotation of a vector by radians

```
rot: vec -> float -> vec
```

and wrote a small program to test these functions. Convert your earlier programs into a library called `Vec`, consisting of a signature file, an implementation file, and an application file. Create a `.fsproj` project file, and run the code using first `dotnet fsi` and then `dotnet run`.

5ø1 A linked list is an abstract datatype that is built into `F#`.

In this exercise you will create your own implementations of a linked list, one only supporting `int` and a generic linked list that can be used with any datatype, just like the built-in `F#` lists.

(a) Copy the following signature into `intLinkedList.fsi` and write a corresponding implementation in `intLinkedList.fs`

```
module IntLinkedList
type intLinkedList = Nil | Cons of int * intLinkedList
val head : intLinkedList -> int
val tail : intLinkedList -> intLinkedList
val isEmpty : intLinkedList -> bool
val length : intLinkedList -> int
val add : int -> intLinkedList -> intLinkedList
```

Write a small test program to ensure your implementation works as expected. You can take inspiration from the following listing:

```
open IntLinkedList
let emptyList = Nil
let l1 = Cons (1, Nil)
let l2 = Cons (1, Cons (2, Nil))
let l3 = add 3 l2
isEmpty emptyList |> printfn "Empty list is empty: %A"
isEmpty l1 |> not |> printfn "Non-empty list is not empty: %A"
head l1 = 1 |> printfn "head gives the first element: %A"
tail l1 = Nil |> printfn "tail gives the rest of the list: %A"
length l3 = 3 |> printfn "l3 has length 3: %A"
```

(b) In the previous sub-exercise the linked list is restricted to the type `int`. In this sub-exercise you should construct a generic linked list module.

Copy and finish the following signature in `linkedList.fsi` and write a corresponding implementation in `linkedList.fs`.

```
module LinkedList
type LinkedList<'a> = Nil | Cons of 'a * intLinkedList<'a>
val head : LinkedList<'a> -> 'a
val tail : ?? // Fill in yourself
val isEmpty : ?? // Fill in yourself
val length :  ?? // Fill in yourself
val add :  ?? // Fill in yourself
```

Write a small test-program showing you can construct linked lists of all types. You should be able to reuse `all` of your test from the previous sub-exercise and thus create linked lists of `LinkedList<int>`, as well as the following:

```
open LinkedList
let emptyList = Nil
let l1Float = add 2.0 emptyList |> add 3.14 // A float list
let l1String = add "Linked lists are cool!" emptyList
let l2String = add "What is cool?" l1String
// A list of int lists
let intLstLst = add l1 emptyList |> add l2 |> add emptyList
// a list of string lists
let strLstLst = add l1String emptyList |> add l2String
```

(c) Implement `fold` for your linked list module, similar to `List.fold`.

(d) Implement `foldBack` for your linked list module, similar to `List.foldBack`.

(e) Implement `map` for your linked list module, similar to `List.map`.

Afleveringsopgaver (in English)

In the following, you are to work with the abstract datatype known as a *queue*. A queue is a a sequence of elements that supports the following operations: checking whether the sequence is empty; removing an element from the front ("left"); adding an element at the end ("right"). Queues appear often in real life: The line[1] waiting for service at a shop counter, orders to be filled in a warehouse, students waiting to be examined at an oral examination.

The abstract datatype of *purely functional queues* consists of:

- the types named `element` and `queue`, where `element` is the type of elements and `queue` the type of queues with such elements;
- the following value and two functions

```
// the empty queue
emptyQueue: queue
// add an element at the end of a queue
enqueue: element -> queue -> queue
// remove and return the element at the front of a queue
// precondition: input queue is not empty
dequeue: queue -> element * queue
// check if a queue is empty
isEmpty: queue -> bool
```

- the properties these operations must satisfy and that another programmer can rely on, such as queuing an element on an empty queue and then dequeuing from it yields the element added at first and leaves an empty queue behind; or, more generally, first repeatedly queuing elements and then dequeuing until the queue is empty yields the same elements.[2]

[1] In American English. Called indeed *queue* in British English.

[2] This can be expressed as a precise mathematical property, which in turn can be used to systematically test one's implementation to find errors. We'll do this only informally here.

These queues are called *(purely) functional* because the enqueue and dequeue operations return a *new* queue whenever they are called, without destroying the old queue. For example, adding an element e_1 to a queue q_0 of length 15 results in a queue q_1 of length 16; then adding another element e_2 to q_0 also results in a queue q_2 of length 16, but one that is different from q_1 in its last element. At this point we have 3 separate queues, each of which we can used in future operations: q_0, q_1 and q_2.[3]

In this exercise, you will work with functional queues in F#. We'll omit writing "functional" below.

5i0 In the following you are to create a dotnet project for the assignment.

(a) Using the `dotnet` command line tool, create a new `F#` console application named `5i`.
`dotnet new console -lang "F#" -o 5i`

(b) Rename the file `Program.fs` to `testQueues.fs`.

(c) Update `5i.fsproj` to compile `testQueues.fs` instead of `Program.fs`.

(d) Ensure the project works by running `dotnet build` and `dotnet run`.

5i1 In the following, you are to implement your own queue module using lists in F# to represent the (sequence of elements in) a queue. The module is to be called `IntQueue`.

(a) Given the description of the abstract datatype Queue above, write a signature file `intQueue.fsi` for the functional queues.

(b) Write an implementation file `intQueue.fs`, implementing the signature file above using lists in F# where the elements are F# integers.

(c) Add `intQueue.fsi` and `intQueue.fs` to `5i.fsproj`,
so they are compiled *before* `testQueues.fs`.

(d) In `testQueues.fs`, show your implementation works by using your queue. As a minimum, you should add the following series of tests of your `IntQueue` module:

```
let intQueueTests () =
    let q0 = IntQueue.emptyQueue
    let emptyTestResult = IntQueue.isEmpty q0
    emptyTestResult
    |> printfn "An empty queue is empty: %A"

    let e1,e2,e3 = 1,2,3
    let q1 = q0 |> IntQueue.enqueue e1
                |> IntQueue.enqueue e2
                |> IntQueue.enqueue e3
    let nonEmptyTestResult = IntQueue.isEmpty q1

    nonEmptyTestResult
    |> printfn "A queue with elements is not empty: %A"

    let (e,q2) = IntQueue.dequeue q1
    let dequeueTestResult = e = e1
    dequeueTestResult
```

[3] There are also *ephemeral* (also called *imperative*) queues, where enqueue and replace the original queue with the new queue such that there is always just one "current" queue that changes over time. Ephemeral queues have more limited functionality and are easier to implement efficiently using imperative data structures, which we will encounter later in the course.

```
    |> printfn "First in is first out: %A"

    let allTestResults =
        emptyTestResult &&
        nonEmptyTestResult &&
        dequeueTestResult

    allTestResults
    |> printfn "All IntQueue tests passed: %A"
    // Return the test results as a boolean
    allTestResults

// Run the IntQueue tests
let intQueueTestResults = intQueueTests ()
```

5i2 A problem with the queue specification above is that there is a precondition on the dequeue operation: A programmer must always ensure that the argument to dequeue is nonempty before calling dequeue. In other words, even though the F# type system does not flag it as an error, it *is* an error (by the programmer) to call dequeue with the empty queue.

In the following, you are to implement another version of your queue module using lists in F# to represent the (sequence of elements in) a queue, with error handling. The module is to be called `SafeIntQueue`, the signature file `safeIntQueue.fsi` and the implementation file `safeIntQueue.fs`.

Change the queue specification such that `SafeIntQueue.dequeue` returns an `(element option) * queue` value and remove the precondition. Add the new module to `5i.fsproj` and in `testQueues.fs`, add a corresponding test suite that shows your implementation works; that is, its operations perform queuing and dequeuing, and, additionally `SafeIntQueue.dequeue(emptyQueue)` returns `None`.

5i3 A queue is an abstract datatype, and as such the operations of a queue should not depend on the type of elements. In the following you are to implement a *generic queue*, so it is possible to create queues of any type, e.g. queues of `int`, queues of `float`, queues of `string` or even queues of queue.

The module is to be called `Queue`, the signature file `queue.fsi` and the implementation file `queue.fs`. Add the new module to `5i.fsproj` and in `testQueues.fs`, add a corresponding test suite that shows your implementation works; that is, its operations perform queuing and dequeuing, and additionally that you can build queues of different types with the same generic library. As a minimum, demonstrate queues of `int`, `float` and `string`.

Krav til afleveringen

Afleveringen skal bestå af

- en zip-fil, der hedder `5i.zip`
- en opgavebesvarelse i pdf-format.

Zip-filen skal indeholde:

- filen `README.txt` som er en tekstfil indeholdende jeres navn og en beskrivelse af hvordan man bygger projektet og kører jeres test.
- en `src` mappe med følgende og kun følgende filer:

  5i.fsproj,
  intQueue.fsi, intQueue.fs,
  safeIntQueue.fsi, safeIntQueue.fs,
  queue.fsi, queue.fs,
  testQueues.fs

  svarende til afleveringsopgaverne. Funktionerne skal være dokumenteret med ifølge dokumentationsstandarden ved brug af `<summary>`, `<param>` og `<returns>` XML tagsne.
- pdf-dokumentet skal være lavet med LaTeX, benytte `opgave.tex` skabelonen, ganske kort dokumentere din løsning og indeholde evt. figurer.

God fornøjelse.