Written exam, Functional Programming
Tuesday 11 June 2013

Version 1.00 of 2013-06-01

These exam questions comprise 7 pages. Check immediately that you have all the pages.

The exam duration is 4 hours.

There are 4 questions. To obtain full marks you must answer all the subquestions satisfactorily.

You are allowed to use books, lecture notes, lecture slides, hand-ins, solutions to assignments, calculators etc. during the examination. You are **not** allowed to use computers, mobile phones, PDA's, iPods, iPads or any other form of device that can execute programs written in F# or C# or Java or Scala or that can communicate with other devices.

If a subquestion requires you to define a particular function, then you may **use that function in subsequent subquestions**, even if you have not managed to define it yourself.

If a subquestion requires you to define a particular function, then you may **define as many helper functions as you want**, but in any case you must define the required function so that it has exactly the type and effect that the subquestion asks for.

Question 1 (25 %)

In this question we will consider a simple cash register, where items are identified by a unique *id*. An item also has a *name* and a unit *price*. This leads to the following declarations:

```
type item = {id : int;
             name : string;
             price : float}

type register = item list
```

Question 1.1

Declare a value of type `register` with the following four items:

1. Item with id 1 named "Milk" with price 8.75
2. Item with id 2 named "Juice" with price 16.25
3. Item with id 3 named "Rye Bread" with price 25.00
4. Item with id 4 named "White Bread" with price 18.50

Question 1.2

Declare an exception `Register` and a function

```
getItemById : int -> register -> item
```

so that `getItemById i r` extracts the first occurrence of an item with id `i` from the given register `r`. The function should raise the exception `Register` if the item is not in the register. The exception should contain an additional string explaining the error.

Question 1.3

Declare a function

```
nextId : register -> int
```

so that `nextId r` returns the next id to use in the register. If *maxId* is the maximum id currently used in register `r`, then the next id is defined as *maxId* + 1. The first id to use on an empty register is 1.

Question 1.4

Declare a function

```
addItem : string -> float -> register -> register
```

so that `addItem n p r` adds a new item with name `n` and unit price `p` to the register `r`. The new item must have the next available id as defined by the function `nextId`.

Question 1.5

Declare a function

```
deleteItemById : int -> register -> register
```

so that `deleteItemById i r` deletes an item with id `i` from register `r` and returns the updated register. The register is unchanged if no item with id `i` exists.

Question 1.6

Declare a function

```
uniqueRegister : register -> bool
```

so that `uniqueRegister r` returns true if all items in the register `r` have unique id's, and returns false otherwise.

Question 1.7

Declare a function

```
itemsInPriceRange : float -> float -> register -> register
```

so that `itemsInPriceRange p d r`, given a price `p`, a delta `d` and a register `r`, returns all items in the register whose prices are within the range $[\mathtt{p} - \mathtt{d}, \ldots, \mathtt{p} + \mathtt{d}]$.
The function returns a new register. You may assume that the delta `d` is positive.

Question 2 (25 %)

Consider the following F# declaration:

```
let rec f n m =
  if m=0 then n
  else n * f (n+1) (m-1)
```

Question 2.1

Give the type of `f` and describe what `f` computes. Your description should focus on what `f` computes, rather than on individual computation steps.

Question 2.2

The function `f` is not tail recursive. Declare a function `f'` that is a tail recursive version of `f`.
Hint: You can either use a continuation function or an accumulating parameter.

Question 2.3

Consider the following F# declaration:

```
let rec z xs ys =
  match (xs, ys) with
    ([],[]) -> []
  | (x::xs,[]) -> (x,x) :: (z xs ys)
  | ([],y::ys) -> (y,y) :: (z xs ys)
  | (x::xs,y::ys) -> (x,y)::(z xs ys)
```

Give the type of `z` and describe what `z` computes. Your description should focus on what `z` computes, rather than on individual computation steps. Give two examples of input and result values that support your description.

Question 2.4

Consider the following F# declaration:

```
let rec s xs ys =
  match (xs,ys) with
    ([],[]) -> []
  | (xs,[]) -> xs
  | ([],ys) -> ys
  | (x::xs,y::ys) -> x::y::s xs ys
```

Give the type of `s` and describe what `s` computes. Your description should focus on what `s` computes, rather than on individual computation steps. Give two examples of input and result values that support your description.

Question 2.5

The function `s` above is not tail recursive. Declare a function `sC` that is a tail recursive version of the function `s`.
Hint: You can use a continuation function.

Question 3 (30 %)

Consider the following F# declarations

```
type Latex<'a> =
    Section of string * 'a * Latex<'a>
  | Subsection of string * 'a * Latex<'a>
  | Text of string * Latex<'a>
  | End

let text1 = Section ("Introduction", None,
              Text ("This is an introduction to ...",
                Subsection ("A subsection", None,
                  Text ("As laid out in the introduction we ...",
                    End))))
```

Question 3.1

What is the type of the declaration `text1` above?

Question 3.2

The type `Latex<'a>` represents simple LaTeX like documents and the value `text1` represents the following text

```
1 Introduction
This is an introduction to ...
1.1 A subsection
As laid out in the introduction we ...
```

Declare a function
`addSecNumbers`
that transforms a value as `text1` above into a new value with section numbers added.
For instance `addSecNumbers text1` must return the following value

```
Section ("Introduction", "1",
  Text ("This is an introduction to ...",
    Subsection ("A subsection", "1.1",
      Text ("As laid out in the introduction we ...",
        End))))
```

For a more complicated example consider the F# declaration

```
let text2 = Section ("Introduction", None,
              Text ("This is an introduction to ...",
                Subsection ("A subsection", None,
                  Text ("As laid out in the introduction we ...",
                    Subsection ("Yet a subsection", None,
                      Section ("And yet a section", None,
                        Subsection ("A subsection more...", None,
                          End)))))))
```

The declaration `text2` represents the following text

```
1 Introduction
This is an introduction to ...
1.1 A subsection
As laid out in the introduction we ...
1.2 Yet a subsection
2 And yet a section
2.1 A subsection more...
```

The function application `addSecNumbers text2` must return the value

```
let text2' = Section ("Introduction", "1",
               Text ("This is an introduction to ...",
                 Subsection ("A subsection", "1.1",
                   Text ("As laid out in the introduction we ...",
                     Subsection ("Yet a subsection", "1.2",
                       Section ("And yet a section", "2",
                         Subsection ("A subsection more...", "2.1",
                           End)))))))
```

Question 3.3

What is the type of the function `addSecNumbers`?

Question 3.4

We now extend the type `Latex` to also include labels and references.

```
type Latex<'a> =
    Section of string * 'a * Latex<'a>
  | Subsection of string * 'a * Latex<'a>
  | Label of string * Latex<'a>
  | Text of string * Latex<'a>
  | Ref of string * Latex<'a>
  | End
```

Consider the following F# declaration

```
let text3 = Section ("Introduction", "1",
              Label("intro.sec",
                Text ("In section",
                  Ref ("subsec.sec",
                    Text (" we describe ...",
                      Subsection ("A subsection", "1.1",
                        Label("subsec.sec",
                          Text ("As laid out in the introduction, Section ",
                            Ref ("intro.sec",
                              Text (" we ...",
                                End))))))))))
```

Declare a function

`buildLabelEnv : LaTeX<'a> -> Map<string,string>`

that given a declaration such as `text3` above returns an environment that maps label names to sections. You can assume that the function `addSecNumbers` has been implemented on the extended version of the type `Latex<'a>`.

The type of the environment is `Map<string, string>`. For the example `text3`, the returned environment contains the following two entries: `"intro.sec"` $\rightarrow$ `"1"` and `"subsec.sec"` $\rightarrow$ `"1.1"`.
Hint: First you call `addSecNumbers` to make sure you have section numbers defined.

Question 3.5

Declare a function

```
toString : Latex<'a> -> string
```

that makes a string representation of the given text as illustrated by the examples above.
You can use the value `nl` defined as

```
let nl : string = System.Environment.NewLine
```

to represent a newline to separate sections and sub-sections from ordinary text. You are not expected to do any form of text formatting, text wrapping etc.
Hint: Your function `toString` can make use of the functions `addSecNumbers` and `buildLabelEnv`.

Question 4 (20 %)

Question 4.1

Consider the F# declaration:

```
let mySeq = Seq.initInfinite (fun i -> if i % 2 = 0 then -i else i)
```

Write the result type and result value of evaluating `Seq.take 10 mySeq`.

Question 4.2

Declare the function

```
finSeq : int -> int -> seq<int>
```

so that `finSeq n M` produces the finite sequence $\mathtt{n}, \mathtt{n}+2, \mathtt{n}+4, \ldots, \mathtt{n}+2\mathtt{M}$.

Question 4.3

Consider the following F# declarations

```
type X = A of int | B of int | C of int * int

let rec zX xs ys =
  match (xs,ys) with
    (A a::aS,B b::bS) -> C(a,b) :: zX aS bS
  | ([],[]) -> []
  | _ -> failwith "Error"

let rec uzX xs =
  match xs with
    C(a,b)::cS -> let (aS,bS) = uzX cS
                  (A a::aS,B b::bS)
  | [] -> ([],[])
  | _ -> failwith "Error"
```

Give the types of the functions `zX` and `uzX`. Describe what the two functions compute. Your description should focus on what is computed, rather than the individual computation steps. Give, for each function, three example input values and result values that support your descriptions.