T-501-FMAL Programming languages, Assignment 1
Spring 2021
Due Fri 12 Feb 2020 at 23:59

Write your solutions in a single .fs file named Assignment1.fs and defining a module `Assignment1`. Follow the template provided; the file must contain your names.

Do not change the names of the functions, their type signatures, do the order of the arguments. Top-level helper functions are ok, you can call them what you want.

F# must process your file without errors (warnings are ok).

1. Code in F# the function `nf : int -> int` defined in standard mathematical notation by:

$$nf(n) = \begin{cases} 1 & \text{if } n < 1 \\ 2 & \text{if } n = 1 \\ 2 * nf(n-1) + 3 * nf(n-2) & \text{if } n > 1 \end{cases}$$

2. (i) Write a recursive function `lastTrue : (int -> bool) -> int -> int` such that `lastTrue f n` is the *largest* integer `i` in the range $0, \ldots, n-1$ such that `f i` is `true`. If there is no such `i`, then `lastTrue f n` should be -1.

   (ii) Using `lastTrue`, write a function
   `lastEqual : 'a -> (int -> 'a) -> int -> int when 'a : equality`
   such that `lastEqual x f n` is the largest `i` in $0, \ldots, n-1$ such that `f i` is equal to `x`. If there is no such `i`, then `lastEqual x f n` should be `-1`.

   (iii) Write a recursive function `firstTrue : (int -> bool) -> int -> int` such that `firstTrue f n` is the *smallest* integer `i` in the range $0, \ldots, n-1$ such that `f i` is `true`. If there is no such `i`, then `firstTrue f n` should be `-1`.

   (iv) If `lastTrue (fun x -> f x > f (x + 1)) 100` evaluates to `-1`, what can you say about `f`? How about if `lastTrue f 100 = firstTrue f 100` is `true`?

3. Write a function `repeat_map : ('a -> 'a) -> 'a list -> 'a list` such that `repeat_map f xs` applies `f` once to the first element of `xs`, twice to the second, three times to the third, and so on.

4. (i) Write directly by recursion a function `sum_some : int option list -> int` that computes the sum of all of the integers in the list.

   (ii) Write the same function using only `fold` by filling in the `...` in

```
let sum_some2 xs =
    List.fold (fun s o ->
        match o with
        ...) 0 xs
```

   (iii) Write the same function using `map` and `fold` by filling in the `...` in

```
let sum_some3 xs =
    let f o = ...
    List.fold (+) 0 (List.map f xs)
```

5. Consider the following datatype of non-empty lists:

```
type 'a nelist =
    | One of 'a
    | Cons of 'a * 'a nelist
```

   The constructor `One` is for singleton lists (`One x` represents the list `[x]`), and `Cons` is the cons constructor.

   (i) Write a function `ne_product : int nelist -> int` that computes the product of all of the elements of a non-empty list.

(ii) Write a function `ne_append : 'a nelist -> 'a nelist -> 'a nelist` that concatenates two non-empty lists.

(iii) Write a function `to_list : 'a nelist -> 'a list` that converts a non-empty list to the corresponding F# list.

(iv) Write a function `ne_map : ('a -> 'b) -> 'a nelist -> 'b nelist` such that the expression `to_list (ne_map f xs) = List.map f (to_list xs)` evaluates to `true`.

(v) Let `to_pair : 'a nelist -> 'a * 'a list` be the function defined by:

```
let to_pair xs =
    match xs with
    | One x -> (x, [])
    | Cons (x, xs) -> (x, to_list xs)
```

Write a function `from_pair : 'a * 'a list -> 'a nelist` such that both of the expressions `to_pair (from_pair (x, xs)) = (x, xs)` and `from_pair (to_pair ys) = ys` evaluate to `true`.

(vi) Is it possible to write a function `from_list : 'a list -> 'a nelist` such that the expressions `to_list (from_list xs) = xs` and `from_list (to_list ys) = ys` evaluate to `true`? Explain why.

6. Consider the following type of trees of integers:

```
type product_tree =
    { value: int
    ; children: product_tree list
    ; product: int option }
```

Each node contains an integer (`value`) and zero or more child nodes (`children`). It also contains an optional `product` field, to store the product of all the values in the subtree rooted by this node locally at the node.

(i) Write a function `are_same : product_tree -> product_tree -> bool` that checks whether the two `product_tree` arguments are equal, except possibly for the `product` fields.

(ii) Write a function `get_product : product_tree -> int` that computes the product of the values in a tree. You should use the `product` fields to avoid performing multiplications as much as possible. Assume that you can trust these fields. I.e., even if this field contains a wrong value, you may use it.

(iii) Write a function `fill_products : product_tree -> product_tree` such that `fill_products t` returns the same tree as `t`, but with all of the `product` fields filled in (so the result should not contain `None`). Again you should avoid unnecessary multiplications as much as possible.