T-501-FMAL Programming languages, Practice class 7
Spring 2021

1. This problem is about using higher-order functions for production of HTML code. This is handy when generating static webpages from database information and when writing Web scripts in a functional style.

   (i) Write an F# function `htmlrow : int * (int -> string) -> string` that builds one row of an HTML table from a the given number of cells in this row and given contents for each cell. For example,

   ```
   htmlrow (4, fun j -> string (j * 8));;
   ```

   should produce a string that will print as

   ```
   <tr><td>0</td><td>8</td><td>16</td><td>24</td></tr>
   ```

   (ii) Write an F# function `htmltable : int * (int -> string) -> string` that builds an HTML table from a given number of rows and the contents of each row. For example,

   ```
   htmltable (3, fun i -> "<tr><td>" + string i + "</td><td>" +
                                                  string (i * 8) + "</td></tr>");;
   ```

   should produce an F# string that will print like this, including line breaks:

   ```
   <table>
   <tr><td>0</td><td>0</td></tr>
   <tr><td>1</td><td>8</td></tr>
   <tr><td>2</td><td>16</td></tr>
   </table>
   ```

   Similarly,

   ```
   htmltable (10,
              fun i -> htmlrow (10,
                                  fun j -> string ((i + 1) * (j + 1))));;
   ```

   should produce a 10-by-10 multiplication table in HTML.
   Newlines are represented by `\n` characters.

2. Run the evaluator from HigherFun.fs on the following expressions given in the concrete syntax. (Of course you first need to convert them into the abstract syntax. You can do this manually.) Is the value of the 3rd one as expected? Explain the result of the 4th expression.

   ```
   let add x = (let f y = x + y in f) in
     add 2 5

   let add x = (let f y = x + y in f) in
     let addtwo = add 2 in
        addtwo 5

   let add x = (let f y = x + y in f) in
     let addtwo = add 2 in
       let x = 77 in
         addtwo 5

   let add x = (let f y = x + y in f) in
     add 2
   ```

3. Use the file HigherFun.fs. Write an F# function `scopeCheck : expr -> unit envir -> unit` that checks that all variables used (also function variables) are defined. It should fail when there is a use of a variable that is not defined (and otherwise just succeed, returning `()`). The environment keeps track of which variables are defined.

```
> scopeCheck (Plus (Var "x", Var "y")) [];;
System.Exception: x not found
> scopeCheck (Plus (Var "x", Var "y")) ["x", (); "y", ()];;
val it : unit = ()
> scopeCheck (Let ("x", Num 5, Plus (Var "x", Var "y"))) ["y", ()];;
val it : unit = ()
```

Hint: What you need to write is quite similar to the type inferrer from FirstFunTypes.fs but simpler (in particular, the type inferrer relied on type declarations in function lets; here this is not needed).

4. Add anonymous functions similar to F#'s `fun x -> e` to the abstract syntax of expressions in HigherFun.fs:

```
type expr =
    | ...
    | Fun of string * expr                // fun x -> e
```

The intent is that, for instance, these two expressions in the concrete syntax

```
fun x -> 2 * x
let y = 22 in fun z -> z + y
```

would be represented in the abstract syntax as

```
Fun ("x", Times (Num 2, Var "x"))
Let ("y", Num 22, Fun ("z", Plus (Var "z", Var "y")))
```

Now extend the evaluation function to cater for the new expression form. Evaluation of an anonymous function should produce a non-recursive closure of the form

```
type value =
    | ...
    | NRF of (string * expr * value envir)
                                          // fun x -> e, [x1,v1;...;xn,vn]
```

In the empty environment, the two expressions shown above should evaluate to

```
NRF ("x", Times (Num 2, Var "x"))
NRF ("z", Plus (Var "z", Var "y"), ["y", I 22])
```

Note that you also have to adjust evaluation of function calls.

5. Experiment with function definitions. Compare the results of evaluating these expressions:

```
let add x = (let f y = x + y in f) in
  add

let add x = fun y -> x + y in
  add

fun x -> fun y -> x + y
```