Assignment 5 (PSD)

PLC: 5.1, 5.7, 6.1, 6.2, 6.3, 6.4 and 6.5

`![hello](./attachments/Pasted image 20231009123328.png)`

Exercise 5.1 The purpose of this exercise is to contrast the F# and Java programming styles, especially as concerns the handling of lists of elements. The exercise asks you to write functions that merge two sorted lists of integers, creating a new sorted list that contains all the elements of the given lists.

(A) Implement an F# function

```
merge : int list * int list -> int list
```

that takes two sorted lists of integers and merges them into a sorted list of integers. For instance, `merge ([3;5;12], [2;3;4;7])` should give `[2;3;3;4;5;7;12]`.

(B) Implement a similar Java (or C#) method

```
static int[] merge(int[] xs, int[] ys)
```

that takes two sorted arrays of ints and merges them into a sorted array of ints. The method should build a new array, and should not modify the given arrays. Two arrays `xs` and `ys` of integers may be built like this:

```
int[] xs = { 3, 5, 12 };
int[] ys = { 2, 3, 4, 7 };
```

Exercise 5.7 Extend the monomorphic type checker to deal with lists. Use the following extra kinds of types:

```
type typ =
  | ...
  | TypL of typ                    (* list, element type is typ *)
  | ...
```

Exercise 6.1 Download and unpack `fun1.zip` and `fun2.zip` and build the micro-ML higher-order evaluator as described in file `README.TXT` point E.

Then run the evaluator on the following four programs. Is the result of the third one as expected? Explain the result of the last one:

```
let add x = let f y = x+y in f end
in add 2 5 end

let add x = let f y = x+y in f end
in let addtwo = add 2
   in addtwo 5 end
end

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

let add x = let f y = x+y in f end
in add 2 end
```

Exercise 6.2 Add anonymous functions, similar to F#'s `fun x -> ...`, to the micro-ML higher-order functional language abstract syntax:

```
type expr =
    ...
  | Fun of string * expr
  | ...
```

For instance, these two expressions in concrete syntax:

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

should parse to these two expressions in abstract syntax:

```
Fun("x", Prim("*", CstI 2, Var "x"))
Let("y", CstI 22, Fun("z", Prim("+", Var "z", Var "y")))
```

Evaluation of a `Fun(...)` should produce a non-recursive closure of the form

```
type value =
  | ...
  | Clos of string * expr * value env   (* (x,body,declEnv) *)
```

In the empty environment the two expressions shown above should evaluate to these two closure values:

```
Clos("x", Prim("*", CstI 2, Var "x"), [])
Clos("z", Prim("+", Var "z", Var "y"), [(y,22)])
```

Extend the evaluator `eval` in file `HigherFun.fs` to interpret such anonymous functions.

Exercise 6.3 Extend the micro-ML lexer and parser specifications in `FunLex.fsl` and `FunPar.fsy` to permit anonymous functions. The concrete syntax may be as in F#: `fun x -> expr` or as in Standard ML: `fn x => expr`, where `x` is a variable. The micro-ML examples from Exercise 6.1 can now be written in these two alternative ways:

```
let add x = fun y -> x+y
in add 2 5 end

let add = fun x -> fun y -> x+y
in add 2 5 end
```

Exercise 6.4 This exercise concerns type rules for ML-polymorphism, as shown in Fig. 6.1.

(i) Build a type rule tree for this micro-ML program (in the let-body, the type of `f` should be polymorphic – why?):

```
let f x = 1
in f f end
```

(ii) Build a type rule tree for this micro-ML program (in the let-body, `f` should *not* be polymorphic – why?):

```
let f x = if x<10 then 42 else f(x+1)
in f 20 end
```

Exercise 6.5 Download `fun2.zip` and build the micro-ML higher-order type inference as described in file `README.TXT` point F.

(1) Use the type inference on the micro-ML programs shown below, and report what type the program has. Some of the type inferences will fail because the programs are not typable in micro-ML; in those cases, explain why the program is not typable:

```
let f x = 1
in f f end

let f g = g g
in f end

let f x =
    let g y = y
    in g false end
in f 42 end

let f x =
    let g y = if true then y else x
    in g false end
in f 42 end

let f x =
    let g y = if true then y else x
    in g false end
in f true end
```

(2) Write micro-ML programs for which the micro-ML type inference report the following types:

- `bool -> bool`
- `int -> int`
- `int -> int -> int`
- `'a -> 'b -> 'a`
- `'a -> 'b -> 'b`
- `('a -> 'b) -> ('b -> 'c) -> ('a -> 'c)`
- `'a -> 'b`
- `'a`

Remember that the type arrow (`->`) is right associative, so `int -> int -> int` is the same as `int -> (int -> int)`, and that the choice of type variables does not matter, so the type scheme `'h -> 'g -> 'h` is the same as `a' -> 'b -> 'a`.