BRPD Assignment 4

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
```

Solution:

The third program returns 7, and not 82, as `x` is already defined in addTwo's closure as 2.

The fourth program returns a closure containing the function `and`, containing a closure containing the variable `x` set to 2. This is due to the fact that `add` is a higher order function, in the sense that it returns a function that receives the second parameter. As the program only sets the first parameter, the result of the program is `add`s inner function.

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.

Solution:

Changes to `Absyn.fs`:
Added following value to expr:

```
   | Fun of string * expr
```

Changes to `HigherFun.fs`:
Added following match option to `eval`:

```
    | Fun(f, fBody) ->
      let funEnv = Closure(f, fBody, env) :: env
      eval fBody funEnv
```

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` , wherex 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
```

Solution:

Changes to `FunLex.fsl`

Added keywords:

```
| "fun"    -> FN1
| "fn"     -> FN2
```

Added tokens for parsing:

```
| "->"            { BODY1 }
| "=>"            { BODY2 }
```

Changes to `FunPar.psy`

Tokens added in the parser

```
%token FN1 FN2 BODY1 BODY2 /* Added for 6.3  */
```

Added the expression to `AtExpr`

```
| FN1 NAME BODY1 Expr                 { Fun($2, $4) }
| FN2 NAME BODY2 Expr                 { Fun($2, $4) }
```

Exercise 6.4

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

Part (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`

Solution:

See Exercise_6_4.png part (i)

Part (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
```

Solution:

See Exercise_6_4.png part (ii)

Exercise 6.5

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

Part (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

val it : string = "int"
```

```
let f g = g g
in f end

error circular
g is of the type a' which is applied a' -> a' ->  .. a'
```

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

type error: bool and int
```

This happens because a type param thats used in an enclosed scope cannot be generalized. (i.e x is bound to int and y has the same type, but y is set to bool interfering with the type.)

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

val it : string = "bool"
```

Part (2)

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

- `bool -> bool`
  - `let f x = if x then true else false in f end`
- `int -> int`
  - `let f x = x+1 in f end`
- `int -> int -> int`
  - ```
    let f x =
        let f2 x2 = x2 + 1 + x in f2 end
    in f end
    ```
- `’a -> ’b -> ’a`
  - ```
    let f x =
       let f2 x2 = x in f2 end
    in f end
    ```
- `’a -> ’b -> ’b`
  - ```
    let f x =
       let f2 x2 = x2 in f2 end
    in f end
    ```
- `(’a -> ’b) -> (’b -> ’c) -> (’a -> ’c)`
  - ```
    let f1 x1 =
      let f2 x2 =
    ```

```
        let f3 x3 =
                x2 (x1 x3)
        in f3 end
   in f2 end
 in f1 end
```

- `’a -> ’b`
    - `let f x = f x in f end`
- `’a`
    - `let f x = f x in f 2 end`

Remember that the type arrow `(->)` is right associative, `soint -> 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`.