The F# Programming Language
CEFP2009 Warm-Up Session (Draft)

Páli Gábor János
E-mail: pgj@elte.hu

Eötvös Loránd University, Faculty of Informatics,
Department of Programming Languages and Compilers

May 22, 2009

Where to Upload Your Solutions

- `https://pnyf.inf.elte.hu/cefp-es/`
- Course: *F# Warm-up*, Exercise: *Session 2*

Mutability

- Use the `mutable` keyword when mutation of data needed.

```
> let mutable variable = "a value";;
val mutable variable : string
> printfn "variable = ’%s’" variable;;
variable = ’a value’
val it : unit = ()
```

- Use left arrow operator (`<-`) to change a value of a “mutable” variable.

```
> variable <- "a new value";;
val it : unit = ()
> printfn "variable = ’%s’" variable;;
variable = ’a new value’
val it : unit = ()
```

Exercise

- Define an `list_add` function that add an element to a list.

  ```
  list_add : 'a list -> 'a -> 'a list
  ```

- Define a mutable `mlist` value and mutate it by `list_add`.

Using References

- Assigning a new value:

```
> let refCell = ref 42;;
val refCell : int ref
> refCell := -1;;
val it : unit = ()
```

- Deferencing:

```
> !refCell;;
val it : int = -1
```

Exercise

- Define the `generateStamp` function by using a reference value.

  ```
  generateStamp : unit -> int
  ```

- It should work like this:

  ```
  > generateStamp ();;
  val it : int = 1

  > generateStamp ();;
  val it : int = 2
  ```

Arranging Code by Modules

- Defining a module:

```
module Settings =
  let version = "1.0.0.0"
  let debugMode = ref false
```

- Using a defined module:

```
module MainProgram
  printfn "Version %s" Settings.version
  open Settings
  debugMode := true
```

Exercise

Use the `Vector2D` record type defined below, and create `length`, `scale`, `shiftX`, `shiftY`, `zero` operations for it in a module.

```
type Vector2D =
  { DX: float; DY: float }
```

Discriminated Unions in Practice

- A sample definition:

```
type Proposition =
  | True
  | And of Proposition * Proposition
  | Or  of Proposition * Proposition
  | Not of Proposition
```

- Traversal of such a type:

```
let rec eval (p: Proposition) =
  match p with
    | True          -> true
    | And(p1,p2)    -> eval p1 && eval p2
    | Or(p1,p2)     -> eval p1 || eval p2
    | Not(p1)       -> not (eval p1)
```

Exercise

- Create an `Ordering` discriminated union that represents the possible results for a comparison: “less than” (`LT`), “equals” (`EQ`), and “greater than” (`GT`).
- Create a `compare` function that compares two elements of the same type and generates an `Ordering`.

```
compare : #System.IComparable
       -> #System.IComparable -> Ordering
```

Example on Using Options

```
let people = [ ("Adam", None)
             ; ("Eve" , None)
             ; ("Abel", Some("Adam", "Eve")) ]


let showParent (name,parents) =
  match parents with
    | Some(dad,mum)
      -> printfn "%s’s parents are %s and %s"
         name dad mum
    | None
      -> printfn "%s has no parents" name
```

Exercise

Define a searching function that searches for an element with a given key in a list, and returns `None` in case of no result.

```
searchByKey : (’a * ’b) list -> ’a -> ’b option
```