Compare Functional Programming in Go with F#

Joshua Andrä

Table of Contents

Introduction

In this term paper I will compare functional programming in F# to funtional programming in Go. At the beginning of this term paper, a short overview over the programming language Syntax of F# will be given. Then there will be a overview of the particularities of F# given. At the end there there will be an explicit example that will be compared.

Syntax

F# is a white space based syntax. That means it is a syntax which is based on new line, indentations and spaces. Compared to Go, where curly braces are required in the body of each function. If the function body of a F# function is longer than one line, the next line hast to be indented.

example.fs

```
let checkIfTeenager x =
  if x >= 13
    then if x <= 19
      then true
```

As can be seen in this example,every new line has been indented instead of being placed in curly bracket blocks. Further, one can see that in a function, the function name and the desired parameters aren't specified.

example.go

```
func add(x int, y int) int {
    return x + y
}
```

As can be see in the Go example the function name and have brackets where one see the parameter with the data type. At the end there is the data type of the return value.

If one compare these two languages one can see that F# doesn't have a variable type assignment. And in Go one give the data types of the return and the variables.

Particularities

Unlike Go, F# is a programming language designed for functional programming, but Go supports the whole functional programming like Lambdaexpressions and closures. One can use the current Go for functional programming but there are some things missing like generics and type inference for function. Also Go uses functional constructs for there standard libary. In functional programming variable types are unchangeable, one can be changed in F# with the keyword "mutable". Even while one can change the variable types in F# it is an functional programming language. Go on the other side is no functional programming language even while one can use Go for functional progrmming.

Lambda and Closure

lambda.fs

```
let operation =
    let state = ref 100
    fun adjust ->
        state := (!state) + adjust
```

lambda.go

```
func intSeq() func() int {
    i := 0
    return func() int {
        i++
        return i
    }
}
```

Both examples aboth show a Lambdaexpression but as one can see in F# it has the keyword "fun" in it which signaled that here starts a Lambdaexpression. Compared to Go one only have the normal func keyword.

Definition of Map, Filter and Reduce

map_filter_reduce.go

```
func Map(vs []string, f func(string) string) []string {
    vsm := make([]string, len(vs))
    for i, v := range vs {
        vsm[i] = f(v)
    }
    return vsm
}

func Filter(vs []string, f func(string) bool) []string {
    vsf := make([]string, 0)
    for _, v := range vs {
        if f(v) {
            vsf = append(vsf, v)
        }
    }
    return vsf

func Reduce(vs []string, f func(string) string) string {
    var vsr string;
    for _, v := range vs {
            vsr = f(vsr) + " " + f(v);
    }
    return vsr
}
```

In the code example for the Map function one can see that there are two given parameters. The first is the array of strings the second calls a function e.g. Uppercase, to print all array entries uppercase.

In the code example from Go for Filter one can see that there is a normal defintion for a function. It takes two parameters one is the string array the other one is the function for the callback.

The example code for Reduce takes two parameters one is the array which holdes the seperated words. The second one is a function for puting the single words together.

map_filter_reduce.fs

```
Array.map : ('T -> 'U) -> 'T [] -> 'U []

Array.filter : ('T -> bool) -> 'T [] -> 'T []

Array.reduce : ('T -> 'T -> 'T) -> 'T [] -> 'T
```

In the codeexample above one can see the definition of map and filter function for an array. The defintion for both function is very short.

In the Map function it says that the array T will be transformed in the array U. In the defenition of the Filter method it says that only the results that give a true boolean value will be returned.

The Reduce method takes an array and puts each element to the string.

If one compare these two defenitions one can see that in Go one write your own Map, Filter and Reduce method in F# one have for each collection function a specific Map, Filter and Reduce function.

Explicit Example

In the following examples one can see the defined function in usage and the return value after calling the function.

map.fs

```
let data = [|"peach"; "apple"; "pear"; "plum"|]
let uppercase (x : string) = x.ToUpper()
let r1 = data |> Array.map uppercase
printfn "%A" r1
```

return

```
[|"PEACH"; "APPLE"; "PEAR";"PLUM"|]
```

map.go

```
var strs = []string{"peach", "apple", "pear", "plum"}
    fmt.Println(Map(strs, strings.ToUpper))
```

return

```
[PEACH APPLE PEAR PLUM]
```

If one compare the two map examples in Go and in F# one can see both return the same array, but for the uppcase letters in F# one are defining a function which can be called every time when one want it. The Syntax of F# make it possible to only write the defined methode behinde the map function because it accepts it as a parameter. In Go one have the typical function call with the given string array and the method for converting the strings into uppercase.

filter.fs

```
let names = [|"peach"; "apple"; "pear"; "plum"|]
let filterE = names |> Array.filter (fun x -> x.Contains("e"))
printfn "%A" filterE
```

return

```
[|"peach"; "apple"; "pear"|]
```

filter.go

```
var strs = []string{"peach", "apple", "pear", "plum"}
fmt.Println(Filter(strs, func(v string) bool {
    return strings.Contains(v, "e")
}))
```

return

```
[peach apple pear]
```

If one compare the filter functions in Go and F# one can see both of them return the values from the array, where an "e" is containg so the Contains function returns true.

In F# one have the "fun" keyword which starts a Lambdaexpression.

In Go one have the typical function call with parameters. The second parameter is a function that returns only the value with boolean value if it is true.

reduce.fs

```
let names = [| "A"; "man"; "landed"; "on"; "the"; "moon" |]
let sentence = names |> Array.reduce (fun acc item -> acc + " " + item)
printfn "%s" sentence
```

return

```
A man landed on the moon
```

reduce.fs

```
 var strs = []string{"A", "man", "landed", "on", "the", "moon"}
    fmt.Println(Reduce(strs, func(v string) string {
        return v
    }))
```

return

```
A man landed on the moon
```

As one can see in the Reduce examples above both return the array elements in a string.

In the F# example there is the "fun" keyword that signaled that there is a Lambdaexpression. The reduce function combines the two parameters acc und item.

In Go one have the Reducemethode which takes the array and returns a string.

Conclusion

If one look at all these examples one can see that the biggest difference between these two languages is the length of each method. Also one can see that F# has explicit keywords for Lambdaexpressions, in go one have the normal function call func. In this examples the Functions for Reduce, Map and Filter are for stringarrays but it is possible to define them in Go for every kind of data type, in F# there is for each collection type a Filter, Reduce and Map function.