Funktionale Programmierung in F# (5)

Parser Combinators

Göran Kirchner[1]

2021-05-03

[1] e_kirchnerg@doz.hwr-berlin.de

Programm

- Hausaufgaben
- Test
- Parser (Kombinatoren)

Accumulate

```fsharp
let rec accumulateR func input acc =
    match input with
    | [] -> acc |> List.rev
    | head::tail -> accumulateR func tail (func head :: acc)
let accumulate func input = accumulateR func input []
let test1 = accumulate (fun x -> x * x) [1; 2; 3]
let test2 = accumulate (fun (x:string) -> x.ToUpper()) ["hello";
↪   "world"]
test2
```

```
val accumulateR: func: ('a -> 'b) -> input: 'a list -> acc: 'b list -> 'b list
val accumulate: func: ('a -> 'b) -> input: 'a list -> 'b list
val test1: int list = [1; 4; 9]
val test2: string list = ["HELLO"; "WORLD"]
val it: string list = ["HELLO"; "WORLD"]
```

Space Age

```
type Planet =
    | Mercury
    | Venus
    | Earth
    | Mars
    | Jupiter
    | Saturn
    | Uranus
    | Neptune
let orbitalPeriodRelativeToEarthOn planet =
    match planet with
    | Mercury -> 0.2408467
    | Venus -> 0.61519726
    | Earth -> 1.0
    | Mars -> 1.8808158
    | Jupiter -> 11.862615
    | Saturn -> 29.447498
    | Uranus -> 84.016846
    | Neptune -> 164.79132
```

Space Age (II)

```
open System
[<Literal>]
let SecondsInOneEarthYear = 31557600.0
let secondsInAYearOn planet =
    SecondsInOneEarthYear * orbitalPeriodRelativeToEarthOn planet
let round (number : float) = Math.Round(number, 2)
let age (planet: Planet) (seconds: int64): float =
    float seconds / (secondsInAYearOn planet)
    |> round
let test1 = age Earth 1000000000L
```

```
[<Literal>]
val SecondsInOneEarthYear: float = 31557600
val secondsInAYearOn: planet: Planet -> float
val round: number: float -> float
val age: planet: Planet -> seconds: int64 -> float
val test1: float = 31.69
```

Zusammenfassung

- nutze exercism.io!
- Formatierung (dotnet fantomas)
- Vermeide `mutable`!!
- nur wichtiges verdient einen Namen
- Vertraue der Pipe (>>, |>, ...)!!
- If-Then-Else mit Boolean ist unnötig
- Parametrisiere!
- If-Then-Else vermeiden ... besser `match`!
- Be lazy! (vermeide `for`-loops)
- Troubleshooting F#
- F#-Styleguide

Test

- 90 Minuten
- Ergebnis per E-Mail an e_kirchnerg@doz.hwr-berlin.de.

⇝ Test

Parser 1 (hard-coded character)

```
open System
let A_Parser str =
    if String.IsNullOrEmpty(str) then
        (false,"")
    else if str.[0] = 'A' then
        let remaining = str.[1..]
        (true,remaining)
    else
        (false,str)
let inputABC = "ABCD";;
let inputZBC = "ZBCD";;
let test11 = A_Parser inputABC
let test12 = A_Parser inputZBC
```

```
val test11: bool * string = (true, "BCD")
val test12: bool * string = (false, "ZBCD")
```

Parser 2 (match a specified character)

```
let pchar (charToMatch,str) =
    if String.IsNullOrEmpty(str) then
        let msg = "No more input"
        (msg,"")
    else
        let first = str.[0]
        if first = charToMatch then
            let remaining = str.[1..]
            let msg = sprintf "Found %c" charToMatch
            (msg,remaining)
        else
            let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch
            ↪  first
            (msg,str)
```

```
val pchar: charToMatch: char * str: string -> string * string
```

Parser 2 (2)

```
let inputABC = "ABCD";;
let inputZBC = "ZBCD";;
let test21 = pchar('A',inputABC)
let test22 = pchar('A',inputZBC)
```

```
val test21: string * string = ("Found A", "BCD")
val test22: string * string = ("Expecting 'A'. Got 'Z'", "ZBCD")
```

Parser 3 (return a Result)

```
let pchar (charToMatch, s) =
    if String.IsNullOrEmpty(s) then
        Error "No more input"
    else
        let first = s.[0]
        if first = charToMatch then
            let remaining = s.[1..]
            Ok (charToMatch, remaining)
        else
            let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch
            ↪  first
            Error msg
```

```
val pchar: charToMatch: char * s: string -> Result<(char * string),string>
```

Parser 3 (2)

```
let test31 = pchar('A',inputABC)
let test32 = pchar('A',inputZBC)
let test33 = pchar('Z',inputZBC)
```

```
val test31: Result<(char * string),string> = Ok ('A', "BCD")
val test32: Result<(char * string),string> = Error "Expecting 'A'. Got 'Z'"
val test33: Result<(char * string),string> = Ok ('Z', "BCD")
```

Parser 4 (use currying)

```
let pchar charToMatch str =
    if String.IsNullOrEmpty(str) then
        Error "No more input"
    else
        let first = str.[0]
        if first = charToMatch then
            let remaining = str.[1..]
            Ok (charToMatch,remaining)
        else
            let msg = sprintf "Expecting '%c'. Got '%c'" charToMatch
            ↪  first
            Error msg
```

```
val pchar: charToMatch: char -> str: string -> Result<(char * string),string>
```

Parser 4 (2)

```
let parseA = pchar 'A'
let inputABC = "ABC"
let inputZBC = "ZBC"
let test41 = parseA inputABC
let test42 = parseA inputZBC
let parseZ = pchar 'Z'
let test43 = parseZ inputZBC
```

```
val parseA: (string -> Result<(char * string),string>)
val inputABC: string = "ABC"
val inputZBC: string = "ZBC"
val test41: Result<(char * string),string> = Ok ('A', "BC")
val test42: Result<(char * string),string> = Error "Expecting 'A'. Got 'Z'"
val parseZ: (string -> Result<(char * string),string>)
val test43: Result<(char * string),string> = Ok ('Z', "BC")
```

Parser 5 (type to wrap the parser function)

```
type Parser<'T> =
    | Parser of (string -> Result<'T , string>)
let pchar charToMatch =
    let innerFn str =
        if String.IsNullOrEmpty(str) then
            Error "No more input"
        else
            let first = str.[0]
            if first = charToMatch then
                let remaining = str.[1..]
                Ok (charToMatch, remaining)
            else
                let msg = sprintf "Expecting '%c'. Got '%c'"
                ↪  charToMatch first
                Error msg
    Parser innerFn
```

```
type Parser<'T> = | Parser of (string -> Result<'T,string>)
val pchar: charToMatch: char -> Parser<char * string>
```

Parser 5 (2)

```
let parseA = pchar 'A'
let inputABC = "ABC"
parseA inputABC
```

```
parseA inputABC;;
  ^^^^^^


...: error FS0003: This value is not a function and cannot be applied.
```

Parser 5 (3)

```
let run parser input =
    let (Parser innerFn) = parser
    innerFn input
let parseA = pchar 'A'
let inputABC = "ABC"
let test1 = run parseA inputABC
let inputZBC = "ZBC"
let test2 = run parseA inputZBC
```

```
val run: parser: Parser<'a> -> input: string -> Result<'a,string>
val parseA: Parser<char * string> = Parser <fun:pchar@74-6>
val inputABC: string = "ABC"
val test1: Result<(char * string),string> = Ok ('A', "BC")
val inputZBC: string = "ZBC"
val test2: Result<(char * string),string> = Error "Expecting 'A'. Got 'Z'"
```

Understanding Parser Combinators

⇝ Understanding parser combinators (Scott Wlashin)

FParsec Tutorial

- FParsec Tutorial
- User's Guide
- FParsec vs alternatives

Using FParsec (1)

```
#r "../src/5/02-fparsec/lib/FParsecCS.dll";;
#r "../src/5/02-fparsec/lib/FParsec.dll";;
open FParsec
let test p str =
    match run p str with
    | Success(result, _, _)   -> printfn "Success: %A" result
    | Failure(errorMsg, _, _) -> printfn "Failure: %s" errorMsg;;
test pfloat "1.25"
test pfloat "1.25E 2"
```

```
Success: 1.25
Failure: Error in Ln: 1 Col: 6
1.25E 2
     ^
Expecting: decimal digit

val it: unit = ()
```

Using FParsec (2)

```
let str s = pstring s
let floatBetweenBrackets:Parser<float, unit>  = str "[" >>. pfloat .>>
↪  str "]";;

test floatBetweenBrackets "[1.0]"
test floatBetweenBrackets "[]"
test floatBetweenBrackets "[1.0]"
```

```
Success: 1.0
Failure: Error in Ln: 1 Col: 2
[]
 ^
Expecting: floating-point number

Success: 1.0
val it: unit = ()
```

Using FParsec (3)

```
let betweenStrings s1 s2 p = str s1 >>. p .>> str s2;;
let floatBetweenBrackets_:Parser<float, unit> = pfloat |>
↪  betweenStrings "[" "]";;
let floatBetweenDoubleBrackets_:Parser<float, unit> = pfloat |>
↪  betweenStrings "[[" "]]";;
test floatBetweenBrackets_ "[1.0]"
test floatBetweenDoubleBrackets_ "[[1.0]]"
let between_ pBegin pEnd p  = pBegin >>. p .>> pEnd;;
let betweenStrings_ s1 s2 p = p |> between_ (str s1) (str s2);;
test (many floatBetweenBrackets) ""
test (many floatBetweenBrackets) "[1.0]"
test (many floatBetweenBrackets) "[2][3][4]"
test (many floatBetweenBrackets) "[1][2.0E]"
```

```
Success: []
Success: [1.0]
Success: [2.0; 3.0; 4.0]
Failure: Error in Ln: 1 Col: 9
[1][2.0E]
        ^

Expecting: decimal digit
```

Zusammenfassung (Kurs)

- Wichtige Werkzeuge (git, dotnet, code)
- Elementare Syntax
- Funktionen, Pattern Matching, Discriminated Unions (DU)
- Tuple, Record, List, Array, Seq
- funktionale Operationen auf Listen (Tail-Rekursion)
- funktionaler Umgang mit fehlenden Daten (Option)
- funktionaler Umgang mit Fehlern (Result)
- funktionales Design (statt Patterns: Funktionen & Verkettung)
- funktionales Refactoring
- funktionales Domain Modeling (DDD)
- eigenschaftsbasiertes Testen (Property Based Testing) (cool!!)
- funktionale Parser (Kombinatoren) (noch cooler!!)

⇝ Was ist Funktionale Programmierung?

Links

- fsharp.org
- docs.microsoft.com/../dotnet/fsharp
- F# weekly
- fsharpforfunandprofit.com
- github.com/../awesome-fsharp

Ende

- Wie geht es weiter?
- Exercism!
- Buchtipps
  - Domain Modeling Made Functional (F#)
  - Stylish F# (F#)
  - Perls of Functional Algorithm Design (Haskell)
  - Thinking Functional with Haskell (Haskell)
  - On Lisp (LISP)
  - Funktionale Programmierung und Metaprogrammierung (LISP)
  - Paradigms of Artificial Intelligence Programming (LISP)
  - Advanced R (R)
- Sprachen: R, Haskell, Clojure, Common Lisp, Elixir, q
- Have FUN!