Functional Languages

Parser combinators - Beautiful parsing,
accompanying material of the parsing lecture and basis for the exercise

17.11.2017

1 Introduction

This document is rather long and includes the most important bits discussed in the lecture. Please note that we will derive a simple parser combinator library which is almost the same as in the lecture but with some stumbling blocks removed... The complete code of this document can be found online `https://tinyurl.com/yasxmkk3` [1].

2 Motivation: Why parser combinators

Parsing strings into data types is a basic utility for computer programmers. For more complex tasks such as XML parsing we often use production quality libraries tuned for specific use cases. For, at first glance, relatively simple tasks however it is a common technique to use string splitting or REGEXes (e.g. command line argument parsing, HTTP URL parsing, user input,. . .). Let us look at the example of METAR reports, the international semi standard format for reporting conditions on airports. Such reports look like:

```
BIRK 281500Z 09014KT CAVOK M03/M06 Q0980 R13/910195
```

Which means:

- For station BIRK
- On 28th of the month
- At 1500 o-clock
- There is wind coming from west (90 degrees) at 14 knots (KT means knots, whereas MTS means meters per second)
- The weather is clear (for code CAVOK)
- The air temperature is -3 degrees celsius
- The dew point is -6 degrees celsius ('M03/M06'), the atmospheric pressure is 0.98kPa ('Q0980')
- Runway 13 has a little frost on it
- But the brakes should bite anyway ("R13/910195")

A rather straightforward approach is to use string splitting in combination with indexing into the right element in order to perform string operations. Let us first focus on the wind part of the METAR report, so let us say we are interested in extracting wind direction and speed. First thing we could do is splitting the METAR string into its components, in order to pick the third item which is the wind part.

```
module AttemptUsingPlainParsing =

    let pMetar1 (metar : string) =
        let manyThings = metar.Split(' ')
        let windPart = manyThings.[2]
        // now parse the wind part. how can we do it.....
        failwith "not implemented, let us build utility functions.."

    // subtask: parse "09014KT"
    let pWindSpeed1 (metarWind : string) =
        // remove prefix
        let knotsString = metarWind.Substring(3)
        // remove knots word
        printfn "%A" knotsString
        let knots = knotsString.Substring(0,knotsString.Length - 2)
        System.Int32.Parse knots

    let testWindSpeed1 = pWindSpeed1 "09014KT" //14
```

Obviously, even with this toy example the code becomes rather cryptic and error prone. In fact our original code could not handle wind speeds measured in MPS.

```
// unfortunately sometimes wind speed is in m/s
let testWindSpeed2 = pWindSpeed1 "09007MPS"
System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number,
 NumberFormatInfo info, Boolean parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
```

Our approach so far is relatively unstructured, and each special case makes our code more complex. The swiss knife of programmers, REGEXes could help right?

```
module Attempt2UsingRegex =
    // ok let us try it with regexes
    let windSpeed2 (metarWind : string) =
        let regex = System.Text.RegularExpressions.Regex("[0-9]{3}([0-9]{2,3})(KT|MPS)")
        let result = regex.Match metarWind
        System.Int32.Parse result.Groups.[1].Value

    // ok
    let testWindSpeed1 = windSpeed2 "09014KT" // 14
    let testWindSpeed2 = windSpeed2 "09007MPS" // 7
```

Although the REGEX works for MPS, we still would need to handle conversions, depending on the unit. Beside that, the code is not really that readable. When REGEXes become more and more complex, they quickly become unmaintainable and hard to read.
Is there a better way to parsing? Which remains readable and maintainable?

3 Entering Parser Combinators (a recap of the lecture)

In this exercise we will implement parser combinators from first principles in order to finally parse METAR strings nicely. Besides from solving our motivational example, parser combinators are a beautiful and practical example of functional programming features such as:

- Higher-order functions and the composition thereof.
- Since parser combinators can be composed freely, they can be seen as a domain specific language for parsing.

In real-world scenarios we would use a full fledged parser combination implementation such as parsec [2]. In fact, the concept carries over to many mainstream programming languages which support functional programming (e.g. nowadays Java or C#). Looking at the core of parser combinators is still a very valuable adventure.

Let us start by defining the parser type (one possible encoding, as mentioned in the lecture).

```
type Input = list<char>
type Parser<'a> = { parse : Input -> Option<'a * Input>} // a record with one field,
                                                             being a function
```

`Input` is a type alias for list¡char¿, which means it is just an abbreviation for writing `list<char>`. Note that, in the lecture we used strings instead of those lists, but for this exercise we use this more precise and efficient implementation. Our parser function returns optionally a parsed value and the remaining input. This is necessary to chain together multiple parsers (as we will see). Next let us define a parsing function, which takes a string input, a parser and returns the parse result.

```
// helpers
let toCharList (s : string) =
    s.ToCharArray() |> Array.toList
let toString (chars : list<char>) =
    String(List.toArray chars)

// takes a parser, a string input and optionally
// returns the parse result and the unconsumed input.
let parse (p : Parser<'a>) (input : string) : Option<'a * Input> =
    p.parse (toCharList input)
```

As discussed in the lecture, we need very primitive parsers (we will need them later on).

```
// creates a parser, which consumes no input and produces the provided value.
let pReturn (a : 'a) : Parser<'a> =
    { parse = fun str -> Some(a,str) }

// creates the ever failing parser
let pFailure : Parser<'a> =
    { parse = fun str -> None }
```

One very basic parser is a parser which parses one single character and uses a function to decide whether the character should be accepted. This one is typically called `sat`:

```
let pSat (f : char -> bool) =
    { parse = fun input ->
        match input with
            | x::xs ->
                if f x then Some (x, xs)
                else None
            | [] -> None
    }
```

As an example we can now use this parser to parse single characters:

```
let test1 = parse (pSat (fun c -> c = 'a')) "a" // all input consumed
let test2 = parse (pSat (fun c -> c = 'a')) "ab" // ok, b remains
let test3 = parse (pSat (fun c -> c = 'a')) "cc"  // no parse result
```

The parser combinator library has 2 tricky functions including this one for chaining together parses:

```
// uses p to parse the input. if p returns a result, use the function to create the next
// parser. handle over the remaining input of p to the parser returned by the function.
let pThen (p : Parser<'a>) (f : 'a -> Parser<'b>) : Parser<'b> =
    { parse = fun input ->
        match p.parse input with
        | None -> None // the first parser failed, all is lost
        | Some(a,rest) ->
            let secondOne = f a // use f to get the next parser
            secondOne.parse rest // continue in secondOne using the remaining input of p
    }
```

Given this parser we can chain together our character parsers:

```
let test4 =
    let parser =
        let a = pSat (fun c -> c = 'a')
        let b = pSat (fun c -> c = 'b')
        pThen a (fun aResult -> b)
    parse parser "ab"
~> Some ('b', [])
```

This parser, for the correct input returns the result of the second one. If we want to return the first result as well, we need to chain in an additional parser:

```
let test5 =
    let parser =
        let a = pSat (fun c -> c = 'a')
        let b = pSat (fun c -> c = 'b')
        pThen a (fun aResult -> pThen b (fun bResult -> pReturn (aResult,bResult)))
    parse parser "ab"
~> Some (('a', 'b'), [])
```

This is really tricky but this is just how the API works, and it is great as you will see at later....

The second tricky function is to express choices (e.g. a parser should work for character a or character b).

```
// tries to parse the input using l, if this does not work it tries out the second one (r).
let pChoice (l : Parser<'a>) (r : Parser<'a>) : Parser<'a> =
    { parse = fun input ->
        match l.parse input with // try l
        | None -> r.parse input // try r
        | Some(a,rest) -> Some(a,rest) // ok l worked
    }
```

In most functional languages we can implement an operator which looks like the OR operator:

```
let (<|>) (l : Parser<'a>) (r : Parser<'a) : Parser<'a> = pChoice l r
```

We can now express choices:

```
let test6 =
   let a = pSat (fun c -> c = 'a')
   let b = pSat (fun c -> c = 'b')
   let aOrB = a <|> b
   parse aOrB "b"
~> Some ('b', [])
```

Since working with `pThen` is not convenient, in languages with support for computation expressions or monads, we are provided by a special syntax. In F# we can create a computation expression builder as such:

```
type ParserBuilder() = // defines an OOP class with two members
    member x.Bind(s : Parser<'a>, f : 'a -> Parser<'b>) : Parser<'b> =
        pThen s f

    member x.Return (s : 'a) : Parser<'a> = pReturn s

let parser = ParserBuilder()
```

Our `pThen` example can now be written so:

```
let test7 =
    let a = pSat (fun c -> c = 'a')
    let b = pSat (fun c -> c = 'b')
    let p =
        parser {
            let! a = a // calls bind on parser, which calls thenP
            let! b = b // same
            return (a,b) // calls return on parser, which then calls pReturn
        }
    parse p "ab"
~>  Some (('a', 'b'), [])
```

More information on computation expression builders can be found here: [3]. As a shortcut, just think of `let!` and `return` to be translated to calls of `Bind` and `Return` in the computation expression builder class. The compiler simple translates the previous code into this:

```
let test8 =
    let a = pSat (fun c -> c = 'a')
    let b = pSat (fun c -> c = 'b')
    let p =
        // let! aResult = a
        parser.Bind(a, fun aResult ->
            // let! bResult = b // same
            parser.Bind(b, fun bResult ->
                // return (a,b)
                parser.Return (a,b)
            )
        )
    parse p "ab"
```

References

[1] Parser Combinators for exercise 5. `https://tinyurl.com/y7jkkveo`. (Accessed on 11/17/2017).

[2] Parsec: Monadic parser combinators. `https://hackage.haskell.org/package/parsec`. (Accessed on 11/17/2017).

[3] The 'Computation Expressions' series — F# for fun and profit. `https://fsharpforfunandprofit.com/series/computation-expressions.html`. (Accessed on 11/17/2017).