Programmering og Problemløsning, 2017

Parsing med Højere-Ordens Funktioner

Martin Elsman

Datalogisk Institut
Københavns Universitet
DIKU

1. December, 2017

Parsing med højere-ordens funktioner

Emner for i dag:

1. **Lexing og Parsing:**
   - Omformning af tekstrenge til interne data-strukturer.
   - Der er tekst-baseret data overalt.
   - 7000+ tekstbaserede data-formater.
2. **Lexing.**
   - Omformning af tekststrenge til "tokens" såsom heltal, ord og symboler.
   - Definition af en simpel "tokenizer".
3. **Parsing.**
   - Parsing med parserkombinatorer.
4. **Eksempler**
   - Eks: Parsing af aritmetiske udtryk, f.eks som input til differentieringskode fra tidligere forelæsning.
   - Eks: Parsing af kommandoer til Logo-fortolker fra tidligere forelæsning.

Lexing

Lexing hentyder til en omformning af strømme af karakterer til strømme af "tokens".

Med *tokens* henvises der til værdier såsom keywords, konstanter, som heltal, floating-point værdier og streng-værdier, samt variabler og symboler.

Ofte vil en lexer også håndtere (og evt fjerne) **kommentarer** i input data.

Til simple formål kan passende definere en funktion der omformer en streng *s* til en liste af "tokens" ved at vi specificerer en række karakterer til opdeling af strengen *s*:

```
val tokenize : string -> string -> string list
```

Eksempel brug af funktionen:

```
> tokenize "+-*() " "34+(a-sin x)*pi";;
val it : string list =
  ["34"; "+"; "("; "a"; "-"; "sin"; " "; "x"; ")"; "*"; "pi"]
```

Funktionen `tokenize`

Her følger en definition af funktionen – `pcomb.fs`:

```
let tokenize (cs:string) (s:string) : string list =
  let extract i n ts =
    if n > 0 then s.Substring (i,n) :: ts
    else ts
  let rec loop i n ts =
    if i+n >= String.length s then
      List.rev(extract i n ts)
    else if String.exists (fun c -> c = s.[i+n]) cs then
      loop (i+n+1) 0 (extract (i+n) 1 (extract i n ts))
    else loop i (n+1) ts
  in loop 0 0 []
```

Bemærk:

- Funktionen `loop` akkumulerer en liste af allerede fundne tokens.
- Funktionen gør brug af metoden `Substring` til at udtrække en delstreng af en streng.

Parsing med Parserkombinatorer

Ultimativt er vi interesserede i at omdanne tekstrenge (eller tokens) til interne datastrukturer:

```
type exp =
  | Int of int
  | X                          // Expressions of one variable
  | Plus of exp * exp
  | Minus of exp * exp
  | Sin of exp
```

Utilities

Vi benytter os a data-typen `'a res`:

```
type 'a res = Ok of 'a | No of string
```

Generel definition af parser:

```
type token = string
type 'a p = token list -> ('a * token list) res
```

Definition af nogle simple parsere

Her følger en parser der kan parse en konkret token:

```
let parse_token (x:string) : string p =
  function s::ts -> if s = x then Ok(s,ts) else No x
         | [] -> No x
```

Her er en parser der kan parse heltal:

```
let parse_int : int p =
  function i::ts -> (try Ok(int(i),ts) with
                         _ -> No "integer")
         | [] -> No "integer"
```

Følgende parser accepterer den ”tomme strøm” (end-of-stream):

```
let eos : unit p =
  function [] -> Ok((),[])
         | _ -> No "tokens"
```

Kombinatorer

Parsere kan sammensættes — deraf navnet *parserkombinator*.

Her følger en kombinator der sammensætter parsere *sekventielt*:

```
let (>*>) (p1:'a p) (p2:'b p) : ('a*'b) p =
  fun ts -> match p1 ts with
              | No s -> No s
              | Ok (v1,ts) ->
                match p2 ts with
                  | No s -> No s
                  | Ok (v2,ts) -> Ok ((v1,v2),ts)
```

Følgende funktion kan bruges til at *transformere* en parser:

```
let (>>@) (p:'a p) (f:'a -> 'b) : 'b p =
  fun ts -> match p ts with
              | No s -> No s
              | Ok (v,ts) -> Ok (f v,ts)
```

Eksempel

```
(parse_int >*> parse_int) >>@ (fun (x,y) -> x+y)
```

Nogle afledte kombinatorer

```
let (->>) (p1:'a p) (p2:'b p) : 'b p =
  (p1 >*> p2) >>@ (fun (_,y) -> y)
let (>>-) (p1:'a p) (p2:'b p) : 'a p =
  (p1 >*> p2) >>@ (fun (x,_) -> x)
```

Bemærk: Disse kombinatorer eliminerer parserresultatet på hhv ventre side (->>) og højre side (>>-).

En kombinator for enten-eller

```
let (|||) (p1:'a p) (p2:'a p) : 'a p =
  fun ts -> match p1 ts with
              | Ok(v,ts) -> Ok(v,ts)
              | No s1 ->
                match p2 ts with
                  | Ok(v,ts) -> Ok(v,ts)
                  | No s2 -> No (s1 + " or " + s2)
```

Bemærk: I tilfælde af fejl sammensættes fejl-beskederne.

Eksempel: `(parse_token "Up" ||| parse_token "Down")` accepterer både strengen Up og strengen Down.

En kombinator for optional parsing

```
let (>>?) (p1:'a p) (p2:'b p) : ('a->'b->'a) -> 'a p =
  fun f ts -> match p1 ts with
                | No s1 -> No s1
                | Ok(v1,ts) ->
                  match p2 ts with
                    | Ok(v2,ts) -> Ok(f v1 v2,ts)
                    | No s2 -> Ok(v1,ts)
```

Parserrepetition

Følgende kombinator omformer en parser til en liste-parser:

```
let rec parse_seq (p:'a p) : 'a list p =
  fun x ->
    (((p >>@ (fun e -> [e])) >>? parse_seq p)
     (fun x y -> x@y)) x
```

Et samlet interface – `pcomb.fsi`

```
module PComb

type 'a res = Ok of 'a | No of string
type token = string
type 'a p = token list -> ('a * token list) res

val (>*>)       : 'a p -> 'b p -> ('a*'b) p
val (>>@)       : 'a p -> ('a -> 'b) -> 'b p
val (->>)       : 'a p -> 'b p -> 'b p
val (>>-)       : 'a p -> 'b p -> 'a p
val (|||)       : 'a p -> 'a p -> 'a p
val (>>?)       : 'a p -> 'b p -> ('a->'b->'a) -> 'a p
val eos         : unit p
val tokenize    : string -> string -> token list
val elimWS      : token list -> token list
val parse_int   : int p
val parse_token : token -> token p
val parse_seq   : 'a p -> 'a list p
val run         : 'a p -> token list -> 'a res  // main
```

Eksempel: Udtryksparsing

Givet følgende data-struktur:

```
type exp = Int of int | X | Plus of exp * exp
  | Minus of exp * exp | Sin of exp
```

Her følger en parser defineret med parserkombinatorer – `pcomb_ex_exp.fs`:

```
let parse_par (p:'a p) : 'a p =
  parse_token "(" ->> p >>- parse_token ")"

let rec parse_exp : exp p = fun x ->
  ((parse_int >>@ Int) |||
   (parse_token "x" >>@ (fun _ -> X)) |||
   ((parse_par (parse_bin "+" Plus))) |||
   ((parse_par (parse_bin "-" Minus))) |||
   ((parse_token "sin" ->> parse_exp) >>@ Sin) |||
   ((parse_par parse_exp))) x
and parse_bin op f =
  (parse_exp >>- parse_token op >*> parse_exp) >>@ f
```

Eksempel: Udtryksparsing — forsat

Hovedfunktionen:

```
  let exp_parser s : exp res =
    let tokens = tokenize "() \n+-" s
    let tokens = elimWS tokens          // eliminate white space
    run (parse_exp >>- eos) tokens
```

Kørsel:

```
> let s = "(2-sin(3+x))"
> do printfn "e=%A" (exp_parser s)
e=Ok (Minus (Int 2,Sin (Plus (Int 3,X))))
```

Udvidelser:

- Parsing med precedens-regler...

Eksempel: Skildpaddegrafik

Kommandoparser – `pcomb_ex_logo.fs`:

```
type cmd = SetColor of string  |  Turn of int
   | Move of int  | PenUp  |  PenDown

let parse_col : string p =
  parse_token "r" ||| parse_token "g" |||
  parse_token "b"

let parse_color_cmd : cmd p =
  (parse_token "c" ->> parse_token "(" ->>
   parse_col >>- parse_token ")") >>@ SetColor

let parse_cmd : cmd p =
  ((parse_token "m" ->> parse_int) >>@ Move) |||
  ((parse_token "t" ->> parse_int) >>@ Turn) |||
  (parse_token "u" >>@ (fun _ -> PenUp)) |||
  (parse_token "d" >>@ (fun _ -> PenDown)) |||
  parse_color_cmd
```

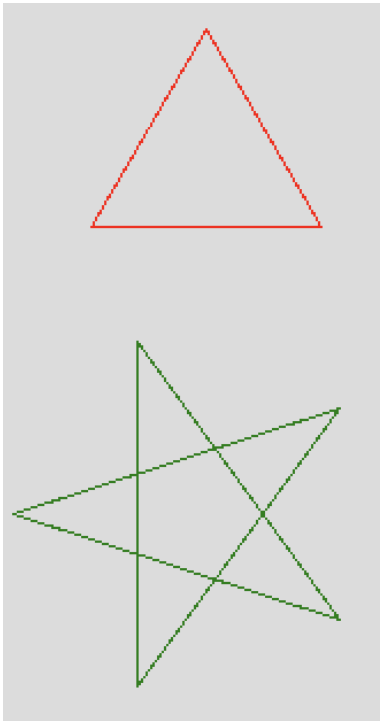
Eksempel: Skildpaddegrafik — forsat

Hovedprogrammet:

```
let logo_parser (s:string) : cmd list res =
  let tokens = tokenize "mtudc#() \n" s
  let tokens = elimWS tokens
  run (parse_seq parse_cmd >>- eos) tokens
```

Kørsel:

```
> let logo = "t70um12c(g)dm65"
> do printfn "%A" (logo_parser logo)
Ok [Turn 70; PenUp; Move 12; SetColor "g";
    PenDown; Move 65]
```



Flere muligheder:

- Tillad gentagelser: `"[m50t144]5"`
- Udvid parser-kombinatorer med bedre fejlbeskeder