AFP8-Documents

December 14, 2024

Objectif.

Savoir faire des requêtes dans des documents structurés (XML/HTML ou JSON par exemple).

1 XML/XPath

Modèle de document

[2]:
```
type Doc =
  | Txt of string
  | Nod of string*list<Doc>

let d = Nod ("person",[Nod ("name",[Txt "john"]); Nod ("surname",[Txt "doe"])])

d |> printfn "%A"
```

```
Nod ("person", [Nod ("name", [Txt "john"]); Nod ("surname", [Txt "doe"])])
```

Sérialisation

[3]:
```
let rec tostr = function
  | Txt t      -> t
  | Nod (n,ds) -> "<"+n+">"+(List.reduce (+) (List.map tostr ds))+"</"+n+">"

d |> tostr |> printfn "%s"
```

```
<person><name>john</name><surname>doe</surname></person>
```

Requêtes simples

[4]:
```
let is n = function
  | Nod (n2,ds) -> if (n=n2) then [Nod (n2,ds)] else []
  | _           -> []

d |> is "person" |> printfn "%A"
```

```
[Nod ("person", [Nod ("name", [Txt "john"]); Nod ("surname", [Txt "doe"])])]
```

[22]:
```
let children = function (** "then" or "followed by" **)
  | Nod (_,ds) -> ds
  | _          -> []
```

```
d |> children |> printfn "%A"
```

```
[Nod ("name", [Txt "john"]); Nod ("surname", [Txt "doe"])]
```

Composition (séquentielle)

[23]:
```
let (>>) q1 q2 n =
  match (q1 n) with
  | [] -> []
  | rs -> List.concat (List.map q2 rs)

d |> (children >> is "name") |> printfn "%A"
```

```
[Nod ("name", [Txt "john"])]
```

Alternatives

[24]:
```
let (<|>) q1 q2 s = (q1 s)@(q2 s)

d |> (children >> (is "name" <|> is "surname")) |> printfn "%A"
```

```
[Nod ("name", [Txt "john"]); Nod ("surname", [Txt "doe"])]
```

Recherche en profondeur (récursivité)

[25]:
```
let deep q s = (q <|> (children >> (deep q))) s

d |> deep (is "name") |> printfn "%A"
```

```
[Nod ("name", [Txt "john"])]
```

Valeurs

[26]:
```
let value = function
  | Nod _ -> []
  | r     -> [r]

d |> (deep (is "name") >> children >> value) |> printfn "%A"
```

```
[Txt "john"]
```

Voir AFP8.fs pour la syntaxe concrète et un exemple d'application.

2 JSON/Mongo

Modèle de document

[1]:
```
(** élément de base **)
type Map<'k,'v> = list<'k*'v>

let rec get k = function
```

```
  | []        -> []
  | (k',v)::m -> if (k'=k) then [v] else get k m

let keys m = List.map (fun (k,_)->k) m

(** structure **)
type E =
  | S of string
  | M of Map<string,E>
  | L of list<E>
```

Exemple

[3]:
```
let e = L [M [("name",S "john");("surname",S "doe")];
           M [("name",S "bob");("surname",S "sponge")]]

let rec prn = function
  | S s -> "'"+s+"'"
  | M m -> m
           |> List.map (fun (k,v)->k+":"+(prn v))
           |> List.reduce (fun x y->x+", "+y)
           |> fun r -> "{ "+r+" }"
  | L l -> l
           |> List.map prn
           |> List.reduce (fun x y->x+", "+y)
           |> fun r -> "[ "+r+" ]"

e
|> prn
|> printfn "%s"
```

```
[ { name:'john', surname:'doe' }, { name:'bob', surname:'sponge' } ]
```

Requête

[7]:
```
let find (M q) = function
  | L l -> l
           |> List.map (fun v->match v with
                               | M m -> let ks = keys q
                                        let vs = List.map (fun k->(get k␣
 ↪q)=(get k m)) ks

                                        let t  = List.reduce (&&) vs
                                        if t then [M m] else []
                               | _   -> [])
           |> List.concat
  | _   -> []

#nowarn "20"
```

```
e
|> find (M [("surname",S "doe")])
|> L |> prn
|> printfn "%s"
```

```
[ { name:'john', surname:'doe' } ]
```