BFNP – Functional Programming

Lecture 8: Text Processing and Sequences

Niels Hallenberg

These slides are based on original slides by Michael R. Hansen, DTU. Thanks!!!

The original slides has been used at a course in functional programming at DTU.

Text Processing - Overview

- Regular Expressions
- TextIO and handling of files
- Full support for culture-dependent information, e.g., sorting
- Conversion to textual format:
  - `sprintf` (formatted string)
  - `printf` (`Console.Out`)
  - `fprintf` (`StreamWriter`)
  - `eprintf` (`Console.Error`)
- XML reader

Regular Expressions (by example)

Consider example string:

```
" John 35 2 Sophie 27 Richard 17 89 3 "
```

First we isolate *person data* strings

```
open System.Text.RegularExpressions
let regOuter = Regex @"\G(\s*[a-zA-Z]+(?:\s+\d+)*)*\s*$"
```

Example:

```
let m1 = regOuter.Match " John 35 2 Sophie 27 Richard 17
89 3 "
captureList m1 1
> val it : string list = [" John 35 2"; " Sophie 27"; "
Richard 17 89 3"]
```

Regular Expressions (by example continued)

We now split each *person data* string and isolate *name* and *data*.
Example:

```
  " John 35 2"
```

```
let regPerson1 =Regex @"\G \s*
  ([a-zA-Z]+) (?:\s+(\d+))*\s*$"
```

Example:

```
let extractPersonData subStr =
  let m = regPerson1.Match subStr
  (captureSingle m 1, List.map int (captureList m 2))

let getData1 str =
  let m = regOuter.Match str
  match (m.Success) with
  | false -> None
  | _     -> Some (List.map extractPersonData (captureList m 1))

getData1 " John 35 2 Sophie 27 Richard 17 89 3 "

> val it : (string * int list) list option =
  Some [("John", [35; 2]); ("Sophie", [27]);
             ("Richard", [17; 89; 3])]
```

Sequences - Overview

- Lazy Lists
- Delayed computations and side–effects
- Cached sequences
- Example: Sieve of Eratosthenes
- Example: Catalogue search
- Type Providers and Databases
- Simple Query Expressions

Sequences (or Lazy Lists)

- *lazy evaluation* or *delayed evaluation* is the technique of delaying a computation until the result of the computation is needed.

Default in lazy languages like Haskell

It is occasionally efficient to be lazy.

A special form of this is *Sequences*, where the elements are not evaluated until their values are required by the rest of the program.

- a *sequence* may be infinite

just a finite part of it is used in computations

Example:

- Consider the sequence of all prime numbers:
  $2, 3, 5, 7, 11, 13, 17, 19, 23, \ldots$
- the first 5 are $2, 3, 5, 7, 11$

Sieve of Eratosthenes

Delayed computations

The computation of the value of *e* can be delayed by ”packing” it into a function (a closure):

```
fun () -> e
```

Example:

```
fun () -> 3+4;;
val it : unit -> int = <fun:clo@10-2>

it();;
val it : int = 7
```

The addition is deferred until the closure is applied.

Example continued

One can make it visible when computations are performed by use of side effects:

```
let idWithPrint i = let _ = printfn "%d" i
                    i;;
val idWithPrint : int -> int

idWithPrint 3;;
3
val it : int = 3
```

The value is printed before it is returned.

```
fun () -> (idWithPrint 3) + (idWithPrint 4);;
val it : unit -> int = <fun:clo@14-3>
```

Nothing is printed yet.

```
it();;
3
4
val it : int = 7
```

Sequences in F#

A lazy list or *sequence* in F# is a possibly infinite, ordered collection of elements, where the elements are computed by demand only.

A natural number sequence $0, 1, 2, \ldots$ is created as follows:

```
let nat = Seq.initInfinite (fun i -> i);;
val nat : seq<int>
```

A `nat` element is computed by demand only:

```
let nat = Seq.initInfinite idWithPrint;;
val nat : seq<int>

Seq.item 4 nat;;
4
val it : int = 4
```

Any type that implements `IEnumerable<'a>` can be used as a sequence.

Further examples

A sequence of even natural numbers is easily obtained:

```
let even = Seq.filter (fun n -> n%2=0) nat;;
val even : seq<int>

Seq.toList(Seq.take 4 even);;
0
1
2
3
4
5
6
val it : int list = [0; 2; 4; 6]
```

Demanding the first 4 even numbers demands a computation of the first 7 natural numbers.

Sieve of Eratosthenes

Greek mathematician (194 – 176 BC)

Computation of prime numbers

- start with the sequence $2, 3, 4, 5, 6, \ldots$
  select head (2), and remove multiples of 2 from the sequence
  2
- next sequence $3, 5, 7, 9, 11, \ldots$
  select head (3), and remove multiples of 3 from the sequence
  $2, 3$
- next sequence $5, 7, 11, 13, 17, \ldots$
  select head (5), and remove multiples of 5 from the sequence
  $2, 3, 5$
- $\vdots$

Sieve of Eratosthenes in F# (I)

Remove multiples of *a* from sequence *sq*:

```
let sift a sq = Seq.filter (fun n -> n % a <> 0) sq;;
val sift : int -> seq<int> -> seq<int>
```

Select head and remove multiples of head from the tail – recursively:

```
let rec sieve sq =
   Seq.delay (fun () ->
                  let p = Seq.item 0 sq
                  Seq.append
                    (Seq.singleton p)
                    (sieve(sift p (Seq.skip 1 sq))));;
val sieve : seq<int> -> seq<int>
```

- Delay is needed to avoid infinite recursion
- Seq.append is the sequence sibling to `@`
- Seq.item 0 sq gives the head of sq
- Seq.skip 1 sq gives the tail of sq

Examples

The sequence of prime numbers and the *n*'th prime number:

```
let primes = sieve(Seq.initInfinite (fun n -> n+2));;
val primes : seq<int>

let nthPrime n = Seq.item n primes;;
val nthPrime : int -> int

nthPrime 100;;
val it : int = 547
```

Re-computation can be avoided by using cached sequences,
`Seq.cache: seq<'a> -> seq<'a>`:

```
let primesCached = Seq.cache primes;;

let nthPrime' n = Seq.item n primesCached;;
val nthPrime' : int -> int
```

Computing the 700'th prime number takes about 8s; a subsequent computation of the 705'th is fast since that computation starts from the 700 prime number

Sieve of Eratosthenes using Sequence Expressions

Sequence expressions can be used for defining step-by-step generation of sequences.

The sieve of Erastothenes:

```
let rec sieve sq =
   seq { let p = Seq.item 0 sq
         yield p
         yield! sieve(sift p (Seq.skip 1 sq)) };;
val sieve : seq<int> -> seq<int>
```

- By construction lazy – no explicit `Seq.delay` is needed
- `yield` *x* adds the element *x* to the generated sequence
- `yield!` *sq* adds the sequence *sq* to the generated sequence
- *seqexp*$_1$
 seqexp$_2$ appends the sequence of *seqexp*$_1$ to that of *seqexp*$_2$

Example: Catalogue search (I)

Extract (recursively) the sequence of all files in a directory:

```
open System.IO ;;

let rec allFiles dir =
  seq {yield! Directory.GetFiles dir
       yield! Seq.collect allFiles (Directory.GetDirectories dir)}
val allFiles : string -> seq<string>
```

where

`Seq.collect: ('a -> seq<'c>) -> seq<'a> -> seq<'c>`

combines a 'map' and 'concatenate' functionality.

```
Directory.SetCurrentDirectory @"C:\mrh\Forskning\Cambridge\";;
let files = allFiles ".";;
val files : seq<string>

Seq.item 100 files;;
val it : string = ".\BOOK\Satisfiability.fs"
```

Nothing is computed beyond element 100.

Example: Catalogue search (II)

We want to search for files with certain extensions, e.g. as follows:

```
let funFiles=Seq.cache (searchFiles (allFiles ".") ["fs";"fsi"]);;
val funFiles : seq<string * string * string>

Seq.item 0 funFiles;;
val it: string * string * string= (".\", "CatalogueSearch", "fs")

Seq.item 6 funFiles;;
val it : string * string * string = (".\BOOK\", "Curve", "fsi")

Seq.item 11 funFiles;;
val it : string * string * string
       = (".\BOOK\", "Satisfiability", "fs")
```

- a sequence in chosen so that the search is terminated when the wanted file is found
- a cached sequence in chosen to avoid re-computation

Example: Catalogue search (III)

The search function can be declared using regular expressions:

```
open System.Text.RegularExpressions ;;

let rec searchFiles files exts =
    let reExts = List.foldBack (fun ext re -> ext+"|"+re) exts ""
    let re = Regex (@"\G(\S*\\)([^\\]+)\.(" + reExts + ")$")
    seq {for fn in files do
            let m = re.Match fn
            if m.Success
            then let path = captureSingle m 1
                 let name = captureSingle m 2
                 let ext  = captureSingle m 3
                 yield (path, name, ext)      };;
val searchFiles : seq<string> -> string list
                   -> seq<string * string * string>
```

- `reExts` is a regular expression matching the extensions
- The path matches the regular expression `\S*\\`
- The file name matches the regular expression `[^\\]+`
- The function `captureSingle` can extract captured strings

Type Providers and Databases

- Language-Integrated Query (LINQ) gives query support and return values of type `IEnumerable<T>` (i.e., sequences)
- A *type provider* for SQL makes the database integration type safe. We use `Sqlite` as an example.

```
type sql = SqlDataProvider<
              ConnectionString = connectionString,
              DatabaseVendor = Common.DatabaseProviderTypes.SQ
              ResolutionPath = resolutionPath,
              IndividualsAmount = 1000,
              UseOptionTypes = true >
```

Say we have two tables `Part` and `PartsList`

```
Part:                          PartsList:
PartId | PartName     PartsListId | PartId | Quantity
  0    |   "Part0"         2      |    0   |     5
  1    |   "Part1"         2      |    1   |     4
  2    |   "Part2"         3      |    1   |     3
  3    |   "Part3"         3      |    2   |     4
```

Type provider and table access

```
let db = sql.GetDataContext()

let partTable = db.Main.Part
val partTable : SqlDataProvider<...>.dataContext.mainSchema.main.P

let partsListTable = db.Main.PartsList
val partsListTable :  SqlDataProvider<...>.dataContext.mainSchema..
```

We can now use the tables as sequences:

```
let r = Seq.item 2 partTable
val r : SqlDataProvider<...>.dataContext.main.PartEntity

r.PartId;;
val it : int = 2

r.PartName;;
val it : string = "Part2"
```

Simple Query Expressions

```
let q1 = query { for part in db.Main.Part do
 select (part.PartName) }
```

returns a sequence with all part names in the table `Part`.
We can join tables:

```
let q2 = query {for pl in db.Main.PartsList do
join part in db.Main.Part on
  (pl.PartsListId = part.PartId)
 select (part.PartName, pl.PartId, pl.Quantity) }
```

We can aggregate:

```
let nextId() = query {for part in db.Main.Part do
count };;
val nextId : unit -> int
```

```
let getDesc id =
  query {for part in db.Main.Part do
  where (part.PartId=id)
  select (part.PartName)
  exactlyOne };;
val getDesc : int -> string
```

Active Patterns

Source: `https://msdn.microsoft.com/en-us/library/dd233248.aspx`

Active patterns makes it possible to decompose data into customized partitions. Data is subdivided into partitions which you name. These names can we used in pattern matching.

```
let (|Even|Odd|) input =
  if input % 2 = 0 then Even else Odd

let TestNumber input =
   match input with
   | Even -> printfn "%d is even" input
   | Odd -> printfn "%d is odd" input

TestNumber 7
TestNumber 11
TestNumber 32
```

Partial Active Patterns, part I

Source: `http://fsharpforfunandprofit.com/posts/convenience-active-patterns/`

Active patterns that do not always produce a value are called *partial active patterns*; they have a return value that is an option type.

```
let (|Int|_|) str =
   match System.Int32.TryParse(str) with
   | (true,i) -> Some i
   | _ -> None

let (|Bool|_|) str =
   match System.Boolean.TryParse(str) with
   | (true,b) -> Some b
   | _ -> None
```

Partial Active Patterns, part II

```
let testParse str =
    match str with
    | Int i -> printfn "The value is an int '%i'" i
    | Bool b -> printfn "The value is a bool '%b'" b
    | _ -> printfn "The value '%s' is something else" str

testParse "12"
testParse "true"
testParse "abc"

> The value is an int '12'
val it : unit = ()
> The value is a bool 'true'
val it : unit = ()
> The value 'abc' is something else
val it : unit = ()
```

Summary

- Anonymous functions `fun () -> ` *e* can be used to delay the computation of *e*.
- Possibly infinite sequences provide natural and useful abstractions
- The computation by demand only is convenient in many applications

It is occasionally efficient to be lazy.

The type `seq<'a>` is a synonym for the .NET type `IEnumerable<'a>`.

Any .NET type that implements this interface can be used as a sequence.

- Lists, arrays and databases, for example.