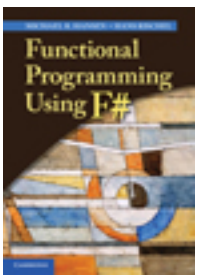


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

10 - Text processing programs pp. 219-250

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.011

Cambridge University Press

10

Text processing programs

Processing text files containing structured data is a common problem in programming – you may just think of analysing any kind of textual data generated by electronic equipment or retrieved data from the web.

In this chapter we show how such programs can be made in a systematic and elegant way using F# and the .NET library. Data are extracted from text files using functions from the `RegularExpressions` library. The data processing of the extracted data is done with a systematic use of F# collections types `list<'a>`, `Map<'a,'b>` and `Set<'a>`. Easy access from F# programs to the extensive text processing features of the .NET library is given in a special `TextProcessing` library that can be copied from the home page of the book. The chapter centers on a real-world example illustrating the techniques.

Time performance of programs is always a problem, even with todays very fast computers. Poor performance of text processing programs is often caused by operations on very long strings. The method in this chapter uses three strategies to avoid using very long strings:

1. Text input is in most cases read and processed in small pieces (one or a few lines).
2. Text is generated and written in small pieces.
3. Large amounts of internal program data are stored in many small pieces in F# collections like `list`, `set` or `map`.

The main focus is on methods for handling textual data both as input and output, but we also illustrate other topics: how to save binary data on the disk to be restored later by another program, and how to read and analyse source files of web-pages. The techniques are illustrated using an example: the generation of a web-page containing a keyword index of the F# and .NET library documentation.

10.1 Keyword index example: Problem statement

Our running example is the generation of a keyword index for the F# and .NET library documentation. The result of this programming effort should be a web-page containing an alphabetically sorted list of keywords, such as the one shown in Figure 10.1. Whenever a user viewing this web-page makes a double click on a keyword, for example, `observer`, a corresponding web-page in the library documentation should automatically be selected by the internet browser.

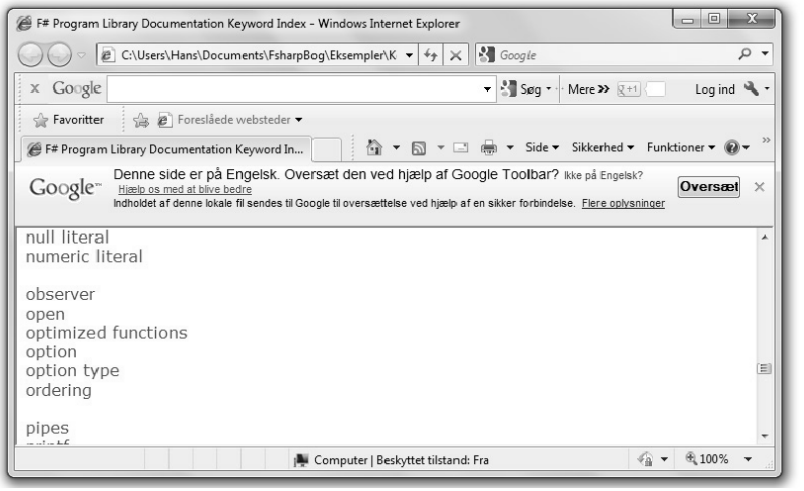


Figure 10.1 A browser's view of `index.html`

```
...
"Control.Observable Module (F#)" observer event~observer
"Control.WebExtensions Module (F#)" async~web~operation
"Microsoft.FSharp.Core Namespace (F#)"
"Core.ExtraTopLevelOperators Module (F#)" top~level~operators
"Core.LanguagePrimitives Module (F#)" language~primitives
"Core.NumericLiterals Module (F#)" numeric~literal
...
```

Table 10.1 *An extract from* `keywords.txt`

The source data to generate the keyword index are found in the `keyword.txt` file that is edited manually by the programmer generating the index (cf. Table 10.1). Each line in this file contains the *title* of a library documentation web-page together with the *keywords* that should refer to this particular web-page. Space characters inside keywords are written using a tilde character such that spaces can be used to separate keywords. The line:

```
"Control.Observable Module (F#)" observer event~observer
```

contains the keywords:

`observer` and `event observer`

(the second containing a space character) with links to the library documentation web-page with title:

```
Control.Observable Module (F#)
```

The programs generating the keyword index from these (and other) data are described in Section 10.8.

10.2 Capturing data using regular expressions

A basic problem in processing textual data is to *capture* the relevant information. In the keyword program we may, for example, input a text line containing a title and two keywords:

```
"Control.Observable Module (F#)" observer event~observer
```

and we want to capture the value

```
("Control.Observable Module (F#)",
        ["observer"; "event observer"])
```

of type `string * (string list)` containing the title and the list of keywords.

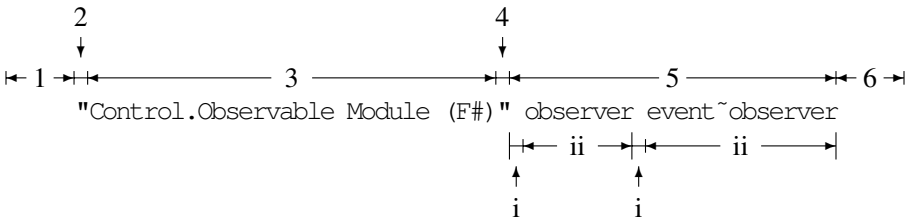
This section presents a systematic technique of constructing functions performing such captures. Using the technique involves three steps

1. An (informal) understanding of the *syntactical structure* of the input.
2. Formalizing this understanding using *regular expressions*.
3. Constructing the function capturing the data.

The difficult part is describing the syntactical structure in terms of regular expressions.

Informal syntactical structure of strings

The syntactic structure of an input line in the keyword index example is illustrated in the following picture using the above line as an example:



We want to capture the parts labelled (3) and (ii) in the figure.

This syntactical structure of an input line can be described by stating that the line should consists of the following parts:

1. Zero or more blank characters.
2. A quote character.
3. One or more non-quote characters (title to be captured).
4. A quote character.
5. Zero or more occurrences of a sequence of.
   i. One or more blank characters.
   ii. One or more non-blank characters (keyword to be captured).
6. Zero or more blank characters.

Construct	Legend
char	Matched by the character *char*. Character *char* must be different from `. $ ^ { [ ( | ) ] } * + ?`
`\`*specialChar*	Matched by *specialChar* in above list (e.g. `$` matches `\$`)
`\`*ddd*	Matched by character with octal value *ddd*
`\S`	Matched by any non-blank character
`\s`	Matched by any blank character
`\w`	Matched by any letter or digit
`\d`	Matched by any decimal digit
`[`*charSet*`]`	Matched by any character in *charSet*
`[^`*charSet*`]`	Matched by any character not in *charSet*
$regExpr_1$ $regExpr_2$	Matched by the concatenation of a string matching $regExpr_1$ and a string matching $regExpr_2$
regExpr `*`	Matched by the concatenation of zero or more strings each matching *regExpr*
regExpr `+`	Matched by the concatenation of one or more strings each matching *regExpr*
regExpr `?`	Matched by the empty string or a string matching *regExpr*
$regExpr_1$ `|` $regExpr_2$	Matched by a string matching $regExpr_1$ or $regExpr_2$
`(?:` *regExpr*`)`	Weird notation for usual bracketing of an expression
`(` *regExpr* `)`	Capturing group
`\G`	The matching must start at the beginning of the string or the specified sub-string (`\G` is not matched to any character)
`$`	The matching must terminate at end of string (`$` is not matched to any character)

charSet = Sequence of chars, char matches and char ranges: $char_1$–$char_2$
The documentation of the `System.Text.RegularExpressions` library contains a link to a regular expression manual.
The F# Power Pack uses another syntax for regular expressions.

Table 10.2 *Selected parts of regular expression syntax*

Regular expressions

Regular expressions formalize the above informal ideas. A regular expression works as a *pattern* for *strings*. Some strings will *match* a regular expression, others will not. We will pay special attention to two kinds of elements in the above informal description:

1. Classes of characters like “a quote character,” “a non-blank character.”
2. Constructs like “sequence of,” “one or more,” “zero or more.”

They are formalized in the regular expression notation as:

1. Single character expressions matched by single characters.
2. Operators for building composite expressions.

Selected parts of the regular expression notation in the .NET library is described in Table 10.2. The upper part of this table contains single character expressions:

- The regular expression `\S` is matched, for example, by the character `P`,
- the regular expression `\d` is matched, for example by the character `5`, and
- the regular expression `\042` is just matched by the character `"`.

The single character expressions `[...]` and `[^...]` are matched by any single character in a set of characters:

- The expression `[ab ]` is matched by any single character among `a`, `b` or *space*, and
- the expression `[^cd]` is matched by any single character except `c` and `d`.

Brackets are used in any algebraic notation whenever an operator is applied to a composite expression like in the expression $(a + b)^2$. In regular expressions we need a further kind of brackets to mark the parts corresponding to data to be captured. There are hence two kinds of brackets in regular expressions:

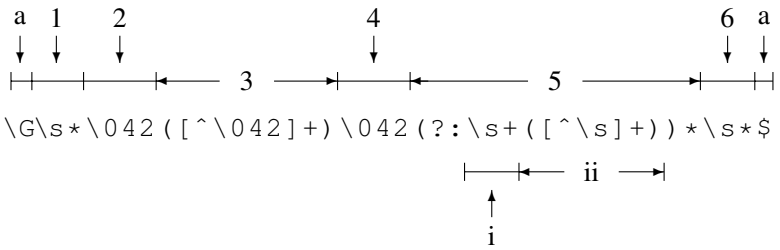
1. *Usual* brackets to enclose a sub-expression when used as operand of an operator.
2. *Capturing* brackets enclosing sub-expressions describing data to be captured.

The designers of the notation have for some mysterious reason decided to use the normal parentheses `(...)` as capturing brackets while the strange notation `(?:...)` is used to denote usual brackets. You just have to accept that the weird symbol `(?:` is the way of writing a usual left bracket in this notation.

Using the notation in Table 10.2 we get the wanted formalization of our description of the syntactical structure of lines in the keyword file in form of the regular expression:

```
\G\s*\042([^\042]+)\042(?:\s+([^\s]+))*\s*$
```

The details in this regular expression can be explained using a picture similar to the previous picture explaining the structure of the string:



The first and last symbols `\G` and `$` labelled (a) are *anchors* used to restrain the matching to all of a string. They are not matched to any characters. The other parts work as follows:

	Expression	Matched by
1	`\s*`	Zero or more blank characters
2	`\042`	A quote character
3	`([^\042]+)`	Capturing group of one or more non-quote characters
4	`\042`	A quote character
5	`(?:...)*`	Zero or more occurrences of:
i	`\s+`	One or more blank characters
ii	`([^\s]+)`	Capturing group of one or more non-blank chars
6	`\s*`	Zero or more blank characters

Name	Type	Legend
`Regex`	`string -> Regex`	Creates regex object for regular expression
regex`.Match`	`string -> Match`	Searches a match to regular expression in string
match`.Length`	`int`	Length of matched string
match`.Success`	`bool`	Result of matching
regex`.Replace`	`string * string -> string`	*regex*`.Replace(`$string_1$`,` $string_2$`)` replaces each match of *regex* in $string_1$ by $string_2$

Table 10.3 *The* `Regex` *class with* `Match` *and* `Replace` *functions*

String matching and data capturing

The `System.Text.RegularExpressions` library contains the types and functions for the .NET regular expressions. A regular expression as the above string is imbedded in a `Regex` object using the `Regex` constructor (cf. Table 10.3):

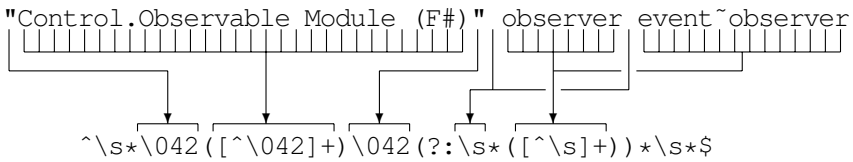
```
open System.Text.RegularExpressions;;
let reg =
    Regex @"\G\s*\042([^\042]+)\042(?:\s+([^\s]+))*\s*$";;
```

We use verbatim `string` constants `@"..."` (cf. Section 2.3) to suppress any conversion of escape sequences because we want to keep all backslash characters unchanged in the string. The expression:

regex`.Match` *string*

evaluates to an object *match* expressing the outcome of matching the string with the regular expression. A `true` value of *match*`.Success` signals a successful matching. The matching will in general try any sub-string of the string but it can be restrained using anchors `\G` and `$` as in the above regular expression where matching only applies to the full string.

The matching of a string to a regular expression determines a mapping from the characters of the string to the single-character expressions of the regular expression. In our example we get the mapping shown in the following picture:



The *capturing groups* in the regular expression are numbered 1, 2, ... according to the order of the opening brackets. In our case we have two capturing groups `([^\042]+)` and `([^\s]+)`. The first is not in the scope of any operator and will hence capture exactly once while the second is in the scope of a `*` operator and may hence capture zero or more times. The picture shows that the second capturing group will capture twice in this case.

The functions `captureSimple` and `captureList` in the `TextProcessing` library of the book (cf. Table 10.4) give a convenient way of extracting the captured data from the `Match` object. The matching and data capture will then proceed as follows:

`captureSingle : Match -> int -> string` `captureSingle` *m n* returns the first captured string of group *n* of the `Match` *m*. Raises an exception if no such captured data.
`captureList : Match -> int -> string list` `captureList` *m n* returns the list of captured strings of group *n* of the `Match` *m*. Raises an exception if the match was unsuccessful.
`captureCount : Match -> int -> int` `captureCount` *m n* returns the number of captures of group *n*
`captureCountList : Match -> int list` `captureCountList` *m* = `[`cnt_0`;` cnt_1`;...;` cnt_k`]` where cnt_n is the number of captures of group *n* (and cnt_0 is some integer).

Table 10.4 *Functions from the* `TextProcessing` *library of the book. See also Appendix B*

```
open TextProcessing;;
let m = reg.Match
    "\"Control.Observable Module (F#)\" observer event~observer";;

m.Success;;
val it : bool = true

captureSingle m 1;;
val it : string = "Control.Observable Module (F#)"

captureList m 2;;
val it : string list = ["observer"; "event~observer"]
```

Conversion of captured data

The data captured as described above are strings. In case of a string of digits (possibly with decimal point) you usually want a *conversion* to values of other types like `int` or `float`. Such conversions can be made using the conversion functions named by the type name like `int` or `float`. Another example of conversion is the capture of a textual representation of a date-time. Such a string may be converted to a `DateTime` value using the `DateTime.Parse` function.

Our example contains a captured title in its final form while the captured keywords need some conversion as any tilde character in the captured string should be replaced by a space character. This conversion is done using the `Replace` function (cf. Table 10.3) in the `Regex` library:

```
let tildeReg = Regex @"~";;
let tildeReplace str = tildeReg.Replace(str," ");;
val tildeReplace : string -> string

tildeReplace "event~observer";;
val it : string = "event observer"
```

Nested data

The `Match` object and function are less elegant in case of nested data like:

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

where we want to capture the data in a form using nested lists:

```
[("John", [35; 2]); ("Sophie", [27]);
                    ("Richard", [17; 89; 3])]
```

The nested syntactic structure is faithfully described in the regular expression

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

with anchors `\G` and `$` enclosing

`(...)*`	Zero or more occurences of capturing group of
`\s*`	Zero or more spaces
`([a-zA-z]+)`	Capturing group of one or more letters
`(?:...)*`	Zero or more occurences of:
`\s+`	One or more spaces
`(\d+)`	Capturing groups of one or more digits
`\s*`	Zero or more spaces

The data groups captured by `Match`:

group 1: `" John 35 2" , " Sophie 27" , " Richard 17 89 3"`
group 2: `"John" , "Sophie" , "Richard"`
group 3: `"35" , "2" , "27" , "17" , "89" , "3"`

can, however, not be used directly to get the above nested list structure – the data captured by group 3 do not reflect the nesting.

A systematic method to capture such data using grammars and parsers is presented in Section 12.10. At present we show two ad hoc ideas to capture the nested data:

- Capture in two steps.
- Using successive calls of `Match`.

Capture in two steps

The two steps are:

1. An "outer" data capture of a list of *person data* strings. In the example it should capture the list `["John 35 2"; " Sophie 27"; " Richard 17 89 3"]` of strings.
2. An "inner" capture of data from each person data string. It should, for example, capture the value `("John", [35; 2])` when applied to the string `"John 35 2"`.

The outer data capture uses the regular expression

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

obtained from the above expression `regNest` by leaving out the two innermost capturing groups. It captures a list of person data strings:

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

The inner data capture uses the regular expression:

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

It captures the person name as a letter string and each integer value as a digit string. The digit strings need further conversions to the corresponding `int` values. This is done using the `List.map` function to apply the conversion function `int` to each digit string:

```
let extractPersonData subStr =
  let m = regPerson1.Match subStr
  (captureSingle m 1, List.map int (captureList m 2));;
val extractPersonData : string -> string * int list
```

Combining these ideas we get the following function:

```
let getData1 str =
  let m = regOuter.Match str
  match (m.Success) with
  | false -> None
  | _     ->
      Some (List.map extractPersonData (captureList m 1));;
val getData1 : string -> (string * int list) list option

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])]
```

Using successive calls of `Match`

The capture is made using successive matches where each match captures from a sub-string containing the data of one person, for example:

sub-string	`" John 35 2 "`	capturing	`("John", [35; 2])`
sub-string	`"Sophie 27 "`	capturing	`("Sophie", [27])`
sub-string	`"Richard 17 89 3 "`	capturing	`("Richard",[17;89;3])`

We use the version of `Match` with a parameter specifying the start position of the sub-string to be matched:

regex`.Match(`*string*`,`*pos*`)`

combined with the regular expression:

```
let regPerson2 =
   Regex @"\G\s*([a-zA-Z]+)(?:\s+(\d+))*\s*";;
```

obtained from the regular expression `regPerson1` by removing the trailing anchor `$`.

Each of the person data sub-strings will then match this regular expression when matching from the *start position* of the sub-string, for instance when matching from the position (=11) of the character "S" in "Sophie":

```
let m =
  regPerson2.Match
    (" John 35 2 Sophie 27 Richard 17 89 3 ", 11);;
captureSingle m 1;;
val it : string = "Sophie"
captureList m 2;;
val it : string list = ["27"]
m.Length ;;
val it : int = 10
```

The length of the captured sub-string is given by `m.Length` and the new position:

$$newPosition = startPosition \texttt{ + m.Length}$$

is the position of the first character "R" in the next person data sub-string.

These are combined in the function `personDataList` that tries to extract a list of person data from the string `str` starting at position `pos` and terminating at position `top`:

```
let rec personDataList str pos top =
  if pos >= top then Some []
  else let m = regPerson2.Match(str,pos)
       match m.Success with
       | false -> None
       | true  -> let data = (captureSingle m 1,
                              List.map int (captureList m 2))
                  let newPos = pos + m.Length
                  match (personDataList str newPos top) with
                  | None     -> None
                  | Some lst -> Some (data :: lst);;
val personDataList : string -> int -> int
                     -> (string * int list) list option
```

The function returns an empty list "`Some []`" if `pos` $\geq$ `top`. Otherwise, a match with the regular expression `regPerson2` is tried. A negative result "`None`" is returned if the match is unsuccessful. Otherwise the data are captured and the new position calculated. The result now depends on the outcome of a recursive call using the new position: A negative result is propagated – otherwise a positive result is obtained by "cons'ing" the captured person data onto the list found in the recursive call.

When applying `personDataList` to a string we start at position `0` with top position equal to the length of the string:

```
let getData2 (s: string) = personDataList s 0 s.Length;;
getData2 " 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])]
```

Several syntactic forms

Use of lines of different syntactic form is frequent: The textual data from a piece of electronic equipment may for instance have the form:

Heading line
Measurement Line
...
Measurement Line
Heading Line
Measurement Line
...

where each heading line initiates a new series of measurements. In this situation one may use one regular expression $regExpr_1$ to describe the syntax of a header line and another expression $regExpr_2$ to describe the syntax of a measurement line. The program analyzing an input line can then use the results *match*`.Success` (cf. Table 10.3) of matching the line against $regExpr_1$ or $regExpr_2$ to get a division into cases.

10.3 Text I/O

Reading and writing text files is done using `File.OpenText` to create a `StreamReader` or using `File.CreateText` to create a `StreamWriter` as described in Table 10.5. These readers and writers are instances of the `TextReader` and `TextWriter` classes that also comprise the `StringReader` and `StringWriter` classes to read and write strings in memory using the same I/O functions (an example of using a `StringReader` is found in Section 10.9. The reader may consult [9] for further details).

The I/O functions insert and remove new-line characters in a proper way. An input line obtained by calling `ReadLine` contains *no* newline character while each call of `WriteLine` adds a new-line at the end of the line. The division into lines inside a text file is hence determined by the successive calls of `WriteLine` plus possible new line characters occurring in the strings written to the file. This structure is retrieved when reading the file line-by-line using successive calls of `ReadLine`.

Name	Type	Legend
`File.OpenText`	`string -> StreamReader`	`File.OpenText(`*path*`)` creates a `StreamReader` to file given by *path*
reader`.ReadLine`	`unit -> string`	Inputs a line from *reader*
reader`.EndOfStream`	`bool`	Indicates no more data in *reader*
reader`.Close`	`unit -> unit`	Closes reading from input medium
`File.CreateText`	`string -> StreamWriter`	`File.CreateText(`*path*`)` creates a `StreamWriter` for writing on a new file as specified by *path*
writer`.WriteLine`	`string -> unit`	Outputs string and newline to *writer*
writer`.Write`	`string -> unit`	Outputs string to *writer*
writer`.Flush`	`unit -> unit`	Flushes buffer of *writer*
writer`.Close`	`unit -> unit`	Closes writing to output medium

Table 10.5 *Selected* `System.IO` *library functions*

A `StreamWriter` has an internal *data buffer*. Part of a string sent to the writer may be temporarily stored in the buffer for later writing to the output medium. A call of `Flush` ensures that the buffer contents is written to the medium.

It is often possible to process an input text on a line-by-line basis. The program will then input one or more lines, do some computations, input the next lines, etc. This pattern of computation is captured in the functions of the `TextProcessing` library of the book described in Table 10.6. Signature and implementation files are given in Appendix B.

A typical application of `fileFold` is to build a collection using a function *f* that captures data from a single input line (as described in Section 10.2) and adds the data to the collection. A typical application of `fileIter` is to generate a side effect for each input line such as output of some data or updating of an imperative data structure. The applications of `fileXfold` and `fileXiter` are similar, but involve several lines of the file.

`fileFold: ('a -> string -> 'a) -> 'a -> string -> 'a` `fileFold` *f e path* = *f* $(\ldots(f\ (f\ e\ lin_0)\ lin_1)\ldots)\ lin_{n-1}$ where $lin_0, lin_1, \ldots, lin_{n-1}$ are the lines in the file given by *path*
`fileIter: (string -> unit) -> string -> unit` `fileIter` *g path* will apply *g* successively to each line in the file given by *path*
`fileXfold: ('a -> StreamReader -> 'a) -> 'a -> string -> 'a` `fileXfold` *f e path* creates a `StreamReader` *rdr* to read from the file given by *path* and makes successive calls *f e rdr* with accumulating parameter *e* until end of the file. Reading from the file is done by *f* using the *rdr* parameter.
`fileXiter: (StreamReader -> unit) -> string -> unit` `fileXiter` *g e path* creates a `StreamReader` *rdr* to read from the file given by *path* and makes successive calls *f rdr* until end of the file. Reading from the file is done by *f* using the *rdr* parameter.

Table 10.6 *File functions of the* `TextProcessing` *library of the book. See also Appendix B*

10.4 File handling. Save and restore values in files

The `File` class of the `System.IO` library contains a large collection of functions to handle files. A few of these functions are described in Table 10.7.

The `Exists` function is useful in a program if you want to ensure that an existing file is not unintentionally overwritten because of errors in the parameters of the program call. The `Replace` function is used in maintaining a file discipline with a current version and a backup version.

The `saveValue` function in the `TextProcessing` library of the book (cf. Table 10.8) can be used to store a value from a program in a disk file. Another program may then later restore the value from the disk file using the `restoreValue` function.

`File.Delete: string -> unit`
  `File.Delete` *path* deletes the file given by *path* (if possible).
`File.Exists: string -> bool`
  `File.Exists` *path* `= true` if a file as specified by *path* exists and `false` otherwise.
`File.Move: string * string -> unit`
  `File.Move(`*oldPath*`,`*newPath*`)` moves file given by *oldPath* to *newPath* (if possible).
`File.Replace: string * string * string -> unit`
  `File.Replace(`*tempPath*`,`*currPath*`,`*backupPath*`)` deletes the file given by *backupPath*, renames the file given by *currPath* to *backupPath* and renames the file given by *tempPath* to *currPath*.

Table 10.7 *Some file handling function from the* `System.IO` *library*

`saveValue: 'a -> string -> unit`
  `saveValue` *value path* saves the value *val* in a disk file as specified by *path*
`restoreValue: string -> 'a`
  `restoreValue` *path* restores a value that has previously been saved in the file given by *path*. Explicit typing of the restored value is required as the data saved in the file do not comprise the F# type of the saved value.

Table 10.8 *Save/restore functions from the books* `TextProcessing` *library. See also Appendix B*

The following examples show how to save two values on the disk:

```
open TextProcessing;;
let v1 = Map.ofList [("a", [1..3]); ("b", [4..10])];;
val v1 : Map<string,int list> =
  map [("a", [1; 2; 3]); ("b", [4; 5; 6; 7; 8; 9; 10])]

saveValue v1 "v1.bin";;
val it : unit = ()

let v2 = [(fun x-> x+3); (fun x -> 2*x*x)];;
val v2 : (int -> int) list = [<fun:v2@20>; <fun:v2@20-1>]

saveValue v2 "v2.bin";;
val it : unit = ()
```

These values are restored as follows:

```
let value1:Map<string,int list> = restoreValue "v1.bin";;
val value1 : Map<string,int list> =
  map [("a", [1; 2; 3]); ("b", [4; 5; 6; 7; 8; 9; 10])]

let [f;g]: (int->int) list = restoreValue "v2.bin";;
val g : (int -> int)
val f : (int -> int)

f 7;;
val it : int = 10

g 2;;
val it : int = 8
```

Note that arbitrary values, including functions, can be saved on the disk and retrieved again and that the type annotations are necessary when restoring the values, because the F# system otherwise would have no information about the types of retrieved values. Furthermore, we have omitted the warning concerning the incomplete pattern `[f;g]` in the last example.

10.5 Reserving, using and disposing resources

A function like `File.OpenText` *reserves other resources* beside allocating a piece of memory for the stream reader object: The disk file is *reserved* for input to avoid any output to the file while it is being used for input. This reservation must be *released* when the program has ceased reading the file.

The release of such resources reserved by functions in the .NET and F# libraries can be managed in a uniform way by using a `use`-binding instead of a `let`-binding when reserving the resources, that is:

`use` *reader* `= File.OpenText` *path*

instead of

`let` *reader* `= File.OpenText` *path*

The keyword `use` indicates to the system that the binding of *reader* comprises resources that should be released once the program is no longer using the object bound to *reader*. The system will in this case release these resources when the binding of the identifier *reader* cannot be accessed any longer from the program. One usually places the `use` declaration inside a function such that the object is automatically released on return from the function.

This mechanism is implemented in the library functions by letting all objects that own resources implement the `IDisposable` interface. This interface contains a `Dispose` operation that is called when the object is released. Declaring `use`-bindings can only be done for such objects.

10.6 Culture-dependent information. String orderings

The .NET library `System.Globalization` offers facilities to handle culture-dependent information like:

- Alphabet with national characters. Ordering of strings, keyboard layout.
- Number printing layout (period or comma). Currency and amounts.
- Date-time format including names of months and days of the week.

Culture-dependent information is collected in a `CultureInfo` object. Such an object is created using the `Name` of the culture, for example

```
open System.Globalization;;
let SpanishArgentina = CultureInfo "es-AR";;
```

The complete collection of the (more than 350) supported cultures is found in the sequence

```
CultureInfo.GetCultures(CultureTypes.AllCultures)
```

and you can get a complete (and long) list of `Name` and `DisplayName` by calling the printing function:

```
let printCultures () =
  Seq.iter
    (fun (a:CultureInfo) ->
          printf "%-12s %s\n" a.Name a.DisplayName)
    (CultureInfo.GetCultures(CultureTypes.AllCultures));;
```

The (mutable) `cultureInfo` object:

```
System.Threading.Thread.CurrentThread.CurrentCulture
```

is used by default in culture-dependent formatting (cf. Section 10.7). The `Name` field:

```
System.Threading.Thread.CurrentThread.CurrentCulture.Name
```

gives the name of the current culture.

Culture-dependent string orderings

The .NET library comprises culture-dependent string orderings. The `orderString` type in the `TextProcessing` library of the book gives a convenient access to these orderings from an F# program (cf. Table 10.9).

Applying the function

```
orderString: string -> string -> orderString
```

to a culture name:

```
let f = orderString cultureName
```

yields a function:

```
f : string -> orderString
```

to create `orderString` values with specified culture,

`orderString: string -> (string -> orderString)`
`orderString` *cultureName*
is a function to create `orderString` values with given culture.
`string: orderString -> string`
`string` *orderString* is the character string contained in *orderString*.
`orderCulture: orderString -> string`
`orderCulture` *orderString* is the culture name of the used string ordering.

Table 10.9 *The* `orderString` *type in the books* `TextProcessing` *library. See also Appendix B*

for example:

```
open System.Globalization;;
open TextProcessing;;

let svString = orderString "sv-SE";;
val svString : (string -> orderString)

let dkString = orderString "da-DK";;
val dkString : (string -> orderString)

let enString = orderString "en-US";;
val enString : (string -> orderString)
```

The comparison operators `compare`, <, <=, > and >= are customized on `orderString` values to the string ordering determined by the culture. We may, for example, observe that the alphabetic order of the national letters ø and å is different in Sweden and Denmark:

```
svString "ø" < svString "å";;
val it : bool = false

dkString "ø" < dkString "å";;
val it : bool = true
```

Comparing `orderString` values with different culture raises an exception:

```
dkString "a" < svString "b";;
... Exception of type 'TextProcessing+StringOrderingMismatch' ...
```

The `string` function gives the string imbedded in an `orderString` value, while the function `orderCulture` gives the culture:

```
let str = svString "abc";;
string str;;
val it : string = "abc"
orderCulture str;;
val it : string = "sv-SE"
```

It is easy to define an `"en-US"` sorting of lists of strings:

```
let enListSort lst =
    List.map string (List.sort (List.map enString lst));;
val enListSort : string list -> string list
```

This function uses `List.map` to apply `enString` to each string in a list of strings. The resulting list of `orderString` values is then sorted using `List.sort`. Finally, the strings are recovered by applying `string` to each element in the sorted list using `List.map`.

The `"en-US"` ordering has interesting properties: Alphabetic order of characters overrules upper/lower case. For example:

```
enListSort ["Ab" ; "ab" ; "AC" ; "ad" ] ;;
val it : string list = ["ab"; "Ab"; "AC"; "ad"]
```

Special characters and digits precede letters:

```
enListSort ["a"; "B"; "3"; "7"; "+"; ";"] ;;
val it : string list = [";"; "+"; "3"; "7"; "a"; "B"]
```

The string ordering corresponds to the order of the entries in a dictionary. This is almost the lexicographical order (ignoring case) – but not quite, for example:

```
enListSort ["multicore";"multi-core";"multic";"multi-"];;
val it : string list
         = ["multi-"; "multic"; "multicore"; "multi-core"]
```

The string "`multicore`" precedes "`multi-core`" because the minus character in this context is considered a hyphen in a hyphenated word, while the string "`multi-`" precedes "`multic`" because the minus character in this context is considered a minus sign, and this character precedes the letter "`c`".

Note the convenient use of a sorting function. One may also obtain textual output sorted according to culture by using values of `orderString` type as keys in `set` or `map` collections: the `fold` and `iter` functions will then traverse the elements of such a collection using the described ordering. The same applies to the enumerator functions of `SortedSet` and `SortedDictionary` collections (cf. Section 8.12).

The ordering of `orderString` values is defined using the `String.Compare` function:

`String.Compare(`$string_1$`,` $string_2$, $cultureInfo$`)`

The user may consult the documentation of this function in [9] for further information about the culture-dependent orderings.

10.7 Conversion to textual form. Date and time

The F# and .NET libraries offer two ways of converting data in a program to textual form: the `sprintf` function with related functions and the `String.Format` function with related functions. The latter is the most advanced, comprising culture-dependent conversion of date or time using a `CultureInfo` object.

The `sprintf` *function*

The `sprintf` function and its fellow functions are called as follows:

`sprintf` *printfFormatString* v_0 ... v_{n-1}
`printf` *printfFormatString* v_0 ... v_{n-1}
`fprintf` *writer printfFormatString* $v_0 \ldots v_{n-1}$
`eprintf` *printfFormatString* v_0 ... v_{n-1}

where *printfFormatString* is a string constant containing a text intermixed with *format placeholders* $fph_0, fph_1, \ldots, fph_{n-1}$ specifying the formatting of the corresponding arguments. The resulting string is obtained from the format string by replacing each format placeholder by result of formatting the corresponding argument.

b	Format boolean value as `true` or `false`
s	String
d, i	Format any basic integer type value as decimal number possibly with sign
u	Format any basic integer type value as unsigned decimal number
e, E	Format floating point value with mantissa and exponent
f, F	Format floating point value in decimal notation
O	Format value using conversion function `string`

Table 10.10 *Some possible format types in a format placeholder*

0	Put zeroes instead of blanks in from of number
-	Left-justify result (within specified width)
+	Use + sign on positive numbers

Table 10.11 *Some possible flags in a format placeholder*

The function `sprintf` delivers the formatted string as the result, while the other functions writes this string on some output media:

`printf`	writes on `Console.Out`.
`fprintf` *writer*	writes on `StreamWriter` *writer*
`eprintf`	writes on `Console.Error`.

A format placeholder has the general form :

`%`{*flags*}{*width*}{`.`*precision*}*formatType*

where {...} means that this part is optional. Frequently used format types and flags are shown in Table 10.10 and Table 10.11.

The integers *width* and *precision* are used in formatting numbers, where *width* specifies the total number of printing positions while *precision* specifies the number of decimals:

```
sprintf "%bhood" (1=2);;
val it : string = "falsehood"
sprintf "%-6d" 67;;
val it : string = "67    "
sprintf "%+8e" 653.27;;
val it : string = "+6.532700e+002"
sprintf "a%+7.2fb" 35.62849;;
val it : string = "a +35.63b"
```

Further information in the documentation of the F# Core Printf Module in [9].

Date and time

A *point of time* is uniquely determined by its universal time (UTC) value (defined by the number of 100 ns ticks since New Year midnight year 1 A.C. at Greenwich). A point in time corresponds to different date-time values at different locations depending on *time zone* and local rules for *daylight saving time*. A standard computer configuration keeps track of local time and UTC time. They are available at any time as the present values of the variables:

```
open System;;
let localNow = DateTime.Now;;     // local time
let UtcNow   = DateTime.UtcNow;; // Utc time
```

It is in general not possible to convert an arbitrary date-time object to universal time.

The `System.TimeZoneInfo` class contains a large array of `TimeZoneInfo` objects

```
let zones = TimeZoneInfo.GetSystemTimeZones()
```

A `TimeZoneInfo` object can be used to convert between local standard time and universal time, but the conversion does not cater for daylight saving time. Further information can be found in

Conversion to textual form using `String.Format`

The function (static member) `Format` of the `String` class in the `System` library has a large number of overloads, in particular:

`String.Format(`*formatString*`,` *values*`)`
`String.Format(`*cultureInfo* `,` *formatString*`,` *values*`)`

where

cultureInfo	A `CultureInfo` object.
formatString	String constant consisting of *fixed texts* intermixed with *format items*.
values	One or several expressions $e_0, e_1, \ldots, e_{n-1}$ separated by comma.

where

fixed text	String not containing any brace characters { or }.
format item	has one of the forms: { *index* } { *index* : *format* } { *index* , *alignment* } { *index* , *alignment* : *format* }

where

index	$k = 0, 1, \ldots$ identifies the argument v_k to be formatted.
alignment	Number of printing positions including heading spaces.
format	One-letter *format code* optionally followed by precision.

`C` or `c`	Currency with national currency symbol
`D` or `d`	Decimal notation
`E` or `e`	Exponential notation
`F` or `f`	Fixed point notation
`N` or `n`	Number
`X` or `x`	Hexadecimal

Table 10.12 *Selected Numeric Format Codes*

d	Short date
D	Long date
t	Short time
T	Long time
F	Long date and long time
g	Short date and short time
M or m	Month and day
Y or y	Year and month

Table 10.13 *Selected Date-time Format Codes*

Precision is an integer between 0 and 99 specifying the number of decimals. There are two kinds of formats: numeric formats as shown in Table 10.12 and date-time formats as shown in Table 10.13. (Further information can be found in the .NET documentation web-pages for `Numeric Format Strings` and `Date Time Format Strings`.)

Some examples:

```
open System ;;
String.Format("{,7:F2}",35.2) ;;
val it : string = "  35,20"
let dk = CultureInfo "da-DK" ;;
let en = CultureInfo "en-US" ;;
let ru = CultureInfo "ru-RU" ;;
let now = DateTime.Now ;;
String.Format(dk, "{1:d}...{0:c}", 45, now) ;;
val it : string = "17-10-2011...kr 45,00"
String.Format(ru,"{0:d}",now) ;;
val it : string = "17.10.2011"
String.Format(en,"{0:d}",now) ;;
val it : string = "10/17/2011"
String.Format(en,"{0:F}",now) ;;
val it : string = "Monday, October 17, 2011 2:57:50 PM"
let ar = CultureInfo "es-AR" ;;
String.Format(ar,"{0:F}",now) ;;
val it : string = "lunes, 17 de octubre de 2011 02:57:50 p.m."
```

10.8 Keyword index example: The `IndexGen` program

We shall now return to the keyword index problem described in Section 10.1. A solution to this problem will be described using the concepts introduced previously in this chapter. We will just describe the main ingredients in the following. The complete program for the solution appears in Appendix A.2. A system diagram for the program generating the keyword web-page, called –IndexGen- is shown in Figure 10.2.

The program reads a keyword file `keywords.txt` and a binary file `webCat.bin` and produces a keyword index web-page `index.html`. An extract from `keywords.txt` is given in Table 10.1. This file is organized into lines where each line contains a title and a list of keywords, like the line containing the title `Control.Observable Module (F#)` and two keywords: `observer` and `event observer`.

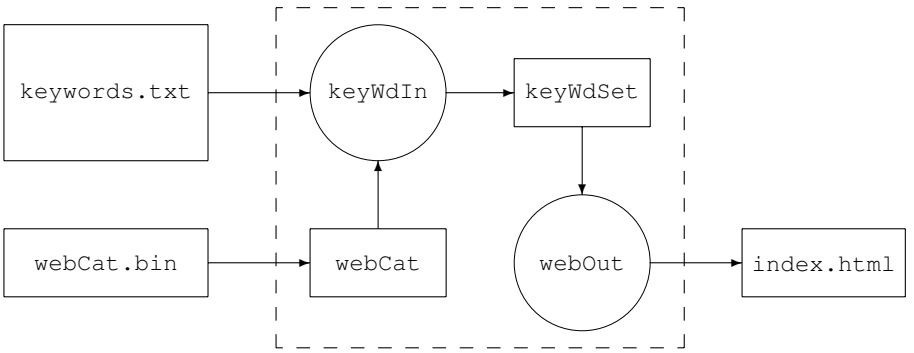


Figure 10.2 System diagram for IndexGen program

The box webCat in the system diagram (see Figure 10.2) is a map from titles to uri's:

```
webCat: Map<string,string>
```

Such a map is called a *webCat map* in the following, and could contain the entry with:

```
key:  "Control.Observable Module (F#)"
value:  "http://msdn.microsoft.com/en-us/library/ee370313"
```

The file webCat.bin is a binary file for a webCat map and we shall see in Section 10.9 how this file to a large extend can be generated automatically.

The set keyWdSet in the system diagram (see Figure 10.2) has the type:

```
keyWdSet: Set<orderString*string>
```

An element in this set is called a *webEntry* and consists of a pair of keyword and associated uri, where the keyword is encoded in an orderString value. The set could include the following two elements:

```
("observer",
        "http://msdn.microsoft.com/en-us/library/ee370313")
("event observer",
        "http://msdn.microsoft.com/en-us/library/ee370313")
```

A set like keyWdSet is called a *webEntry set* in the following.

The set keyWdSet is generated from the files: keyword.txt and webCat.bin by the function: keyWdIn. The index.html file is generated by the webOut function on the basis of keyWdSet. Table 10.14 gives a short extract of the generated web-page and Figure 10.1 shows how the index.html file is shown in a browser. Clicking on the keyword observer in the browser will show the web-page with uri:

```
http://msdn.microsoft.com/en-us/library/ee370313
```

that is, the web-page containing the documentation of observer.

```
...
<a href="http://msdn.microsoft.com/en-us/library/ee353608">
null literal</a><br />
<a href="http://msdn.microsoft.com/en-us/library/ee353820">
numeric literal</a><br />
<br />
<a href="http://msdn.microsoft.com/en-us/library/ee370313">
observer</a><br />
<a href="http://msdn.microsoft.com/en-us/library/ee353721">
open</a><br />
<a href="http://msdn.microsoft.com/en-us/library/ee340450">
...
```

Table 10.14 *An extract of* `index.html`

Data Capture

We now show how to capture the text in our keyword index file (see Section 10.1) to allow *comment lines* in the keyword file. We will allow two types of *comment* lines, a blank line and a line starting with two slash characters.

These two syntactic patterns are described by the regular expression `comReg` containing the "or" operator `|`. Note that the sub-expression `\G//` for comment lines is without trailing anchor "$". A string will hence match this pattern just if the two first characters in the string are slash characters.

```
let comReg = Regex @"(?:\G\s*$)|(?:\G//)";;
```

A normal line with keyword data matches the regular expression:

```
let reg =
  Regex @"\G\s*\042([^\042]+)\042(?:\s+([^\s]+))*\s*$";;
```

The `getData` function should return a result of type:

```
type resType = | KeywData of string * string list
               | Comment
               | SyntError of string;;
```

and the replacement of tilde characters by spaces is made by the function `tildeReplace`:

```
let tildeReg = Regex @"~";;
let tildeReplace str = tildeReg.Replace(str," ");;
```

These ideas and the techniques from Section 10.2 are combined in the function `getData`:

```
let getData str =
    let m = reg.Match str
    if m.Success
    then KeywData(captureSingle m 1,
                  List.map tildeReplace (captureList m 2))
    else let m = comReg.Match str
         if m.Success then Comment
         else SyntError str;;
val getData : string -> resType
```

The function keyWdIn

This part uses the techniques in Section 10.6 to create a webEntry set, where the keywords are ordered using a cultural dependent ordering.

The declarations of keyWdIn is as follows:

```
let enString = orderString "en-US";;
val enString : (string -> orderString)

let keyWdIn() =
  let webCat = restoreValue "webCat.bin"
  let handleLine (keywSet: Set<orderString*string>) str =
    match getData str with
    | Comment          -> keywSet
    | SyntError str    -> failwith ("SyntaxError: " + str)
    | KeywData (title,keywL) ->
      let uri = Map.find title webCat
      let addKeywd kws kw = Set.add (enString kw, uri) kws
      List.fold addKeywd keywSet keywL
  let keyWdSet = Set.empty<orderString*string>
  fileFold handleLine keyWdSet "keywords.txt";;
val keyWdIn : unit -> Set<orderString * string>
```

The idea is to build a webEntry set by folding a function handleLine over all the lines of the keywords.txt file. The function handleLine translates the title in a line to the corresponding uri using the webCat map (that has earlier been input from the webCat.bin file). This uri is then paired with each keyword in the line and these pairs are inserted in the webEntry set.

The function webOut

The function

```
webOut: Set<orderString * string> -> unit
```

that generate the file index.html on the basis of a webEntry set is given in Table 10.15. The file index.html is encoded in the HTML (Hyper Text Mark-up Language) format. The preamble contains the HTML string that sets up the web-page while the postamble contains the HTML-tags that must appear at the end of the file. Please consult Appendix A.1 for a brief introduction to HTML, and Appendix A.2 for the complete source code.

The file index.html is opened and the pre-amble is written on the file. A function outAct to output a single keyword line is declared, and the keywords with uri's in the webEntry set keyWdSet are written on the file using Set.fold. The post-amble is written and the file is closed.

The local outAct function

```
outAct: char -> (orderString * string) -> char
```

outputs one keyword with uri from keyWdSet to the web-page. An *extra empty line* is inserted whenever the first character of the keyword is a letter different from the previous

```
let preamble =  "<!DOCTYPE  ......   /p>";;
let postamble = "</body></html>" ;;

let webOut(keyWdSet) =
  use webPage = File.CreateText "index.html"
  let outAct oldChar (orderKwd: orderString,uri: string) =
    let keyword = string orderKwd
    let newChar = keyword.[0]
    if Char.ToLower newChar <> Char.ToLower oldChar
        && Char.IsLetter newChar
    then webPage.WriteLine "<br />"
    else ()
    webPage.Write "<a href=¨
    webPage.Write uri
    webPage.WriteLine "¨>"
    webPage.Write (HttpUtility.HtmlEncode keyword)
    webPage.WriteLine "</a><br />"
    newChar

  webPage.WriteLine preamble
  Set.fold outAct 'a' keyWdSet |> ignore
  webPage.WriteLine postamble
  webPage.Close()
```

Table 10.15 *The* `webOut` *function*

first character. The argument of the function is, therefore, the *first character* of the *previous keyword* and the value of the function is the first character of the just treated keyword.

The keyword is extracted from the `orderString` value. It becomes a displayed text in the web-page and must hence be encoded in HTML encoding. This is done using the `HttpUtility` function `HtmlEncode` from the `System.Web` library. The uri becomes an attribute and should hence *not* be encoded.

10.9 Keyword index example: Analysis of a web-source

This function addresses the problem of creating a suitable `webCat` map. A major challenge is that the library documentation is spread over a huge number of different web-pages.

The documentation web-pages that are used in the keyword index have the property that they are *organized in two trees*. Any web-page of interest can be reached by following links starting from one of the root pages:

- F# Core Library Reference
- .NET Framework Class Library

These root pages have the following uri's:

- `http://msdn.microsoft.com/en-us/library/ee353567.aspx`
- `http://msdn.microsoft.com/en-us/library/gg145045.aspx`

Scanning the HTML source of the web-page of the F# Core Library Reference we may for instance find the following link to another documentation web-page:

```
<a data-tochassubtree="true"
href="/en-us/library/ee370255"
id="ee340636_VS.100_en-us"
title="System.Collections Namespace (F#)">
System.Collections Namespace (F#)</a>
```

This can be used to find the new title `System.Collections Namespace (F#)` with associated path `/en-us/library/ee370255`. The path is relative to the current web-page but it can be converted to an absolute uri.

Using an XML reader

Analysis of the HTML-source of a web-page can be made using an XML reader, provided that the web-page conforms to the XML-syntax. This is, fortunately, the case for the F# and .NET documentation pages. The *complete* HTML-source of the web-page with given `uri` is read in *one* operation and made available as a string (named `doc`). A `stringReader` is created, and an `XMLReader` is then created to be used in the analysis of the HTML-source:

```
let baseUri = Uri uri
let webCl = new WebClient()
let doc = webCl.DownloadString baseUri
use docRd  = new StringReader(doc)
let settings = XmlReaderSettings(DtdProcessing
                                 = DtdProcessing.Ignore)
use reader = XmlReader.Create(docRd,settings)
```

The `ignore` setting of `DtdProcessing` is required for some security reasons. The XML reader is a mutable data structure pointing at any time to an XML *node* in the HTML-source. Successive calls of *reader*`.Read()` will step the reader through the nodes. Properties of the current node are found as the current values of members of the *reader* object, where we will use the following properties of the current node:

```
reader.NodeType:       XmlNodeType
reader.Name:           string
reader.Value:          string
reader.GetAttribute:   string -> string
reader.Depth:          int
```

We use the following values of `XmlNodeType`:

`XmlNodeType.Element`	HTML element as given by *reader*`.Name`
`XmlNodeType.EndElement`	HTML end element as given by *reader*`.Name`
`XmlNodeType.Text`	Text to be displayed as given by *reader*`.Value`

The library documentation

The data capture from a Microsoft library documentation page uses a specific property that is common for these pages, namely the *navigation menu* shown in the left column of the page. It contains *buttons* like the following:

```
...
Visual F#
F# Core Library Reference
   Microsoft.FSharp Collections Namespace
      Collections.Array Module (F#)
      Collections.Array2D Module (F#)
...
```

The button shown in orange (here in **bold**) points to the current page. It is indented one level. The interesting part (for us) are the buttons just below pointing to the next level of documentation. The start of this sub-menu of the navigation menu is indicated in the web-source by a `div` start element with `class` attribute `toclevel2 children`:

```
<div ... class="toclevel2 children" ...>
```

while the end of the sub-menu is indicated by a matching `div` end element `</div>` at the same level (that is, with the same `Depth`).

A button is given by

```
<a ... href="path" ... >text</a>
```

as described in Appendix A.1.

These observations are captured in the function `nextInfo` in Table 10.16 that steps an `XmlReader` forward to the next "node of interest".

The possible values returned by `nextInfo` indicate:

`StartInfo` d	Found the start `div` element of the sub-menu at depth d.
`EndDiv` d	Found an end `div` element at depth d.
`RefInfo(`*text*`,`*path*`)`	Found a button `<a ...href="`*path*`"...>`*text*`</a>`.
`EndOfFile`	The `XmlReader` has reached the end of the file.

The function `nextInfo` shown in Table 10.16 is the essential ingredient in constructing the function `getWebRefs`

```
val getWEBrefs : string -> (string * string) list
```

that takes a uri as argument and reads through the corresponding web-page source and extracts the list of pairs `(`*title*`,`*uri*`)` corresponding to buttons in the above described sub-menu of the navigation menu. The complete program is found in Appendix A.3. The reader may pay special attention to the following:

- The use of the `HttpUtility.HtmlDecode` function.
- The use of an `Uri` object to convert a *path* in a link to the corresponding absolute *uri*.

```
type infoType =
    | StartInfo of int | EndDiv of int
    | RefInfo of string * string | EndOfFile;;

let rec nextInfo(r:XmlReader) =
    if not (r.Read()) then EndOfFile
    else match r.NodeType with
         | XmlNodeType.Element ->
           match r.Name with
           | "div" when (r.GetAttribute "class"
                         = "toclevel2 children")
                 -> StartInfo (r.Depth)
           | "a" -> let path = r.GetAttribute "href"
                    ignore(r.Read())
                    RefInfo(r.Value,path)
           |  _  -> nextInfo r
         | XmlNodeType.EndElement when r.Name = "div"
                 -> EndDiv (r.Depth)
         |  _    -> nextInfo r;;
val nextInfo: XmlReader -> infoType
```

Table 10.16 *The* `nextInfo` *function*

10.10 Keyword index example: Putting it all together

The keyword index is generated using the text files:

- `webCat0.txt`
- `keywords.txt`

and the programs:

- `NextLevelRefs`
- `MakeWebCat`
- `IndexGen`

The text file `webCat0.txt` is shown in Table 10.17. It contains two pairs of lines with the title and uri of the two root documentation pages. The text file `keywords.txt` contains titles of documentation web-pages with associated keywords and an extract of the file is shown in Table 10.1.

The keyword index is generates in three steps:

1. Generate the text files `webCat1.txt` and `webCat2.txt` using `NextLevelRefs`.
2. Generate the binary file `webCat.bin` using `MakeWebCat`.
3. Generate the text file `index.html` using `IndexGen`.

```
F# Core Library Reference
http://msdn.microsoft.com/en-us/library/ee353567.aspx
.NET Framework Class Library
http://msdn.microsoft.com/en-us/library/gg145045.aspx
```

Table 10.17 *The file* `webCat0.txt`

Generating `webCat1.txt` *and* `webCat2.txt`

This step uses the program `NextLevelRefs` (as shown in Appendix A.3). It operates on *webCat* text files that consists of consecutive pairs of lines containing title and uri for documentation web-pages (like `webCat0.txt`). The program is called with such a file as input. It applies the function `getWebRefs` to each uri in the input file and produces a webCat output file containing all titles and uri's found in this way.

The program is called twice: first with input file `webCat0.txt` to create the new webCat file `webCat1.txt` containing all titles with uri's one level down in the library documentation trees and then with input file `webCat1.txt` to create the new webCat file `webCat2.txt` containing all titles with uri's two levels down in the library documentation trees.

Using a free-standing `NextLevelRefs` program (cf. Section 1.10 and Section 7.2) one makes two calls in a command prompt (assuming that the webCat files and the program are placed in the same directory):

```
NextLevelRefs webCat0.txt webCat1.txt
NextLevelRefs webCat1.txt webCat2.txt
```

Using the file `NextLevelRefs.fsx` in an interactive environment one makes two calls of the `main` function:

```
main [| "webCat0.txt"; "webCat1.txt" |];;
val it : int = 0
main [| "webCat1.txt"; "webCat2.txt" |];;
val it : int = 0
```

where the webCat files are placed in a directory defined by the interactive environment. The keyword index is designed to contain references to documentation web-pages two levels down in the trees, so the files `webCat0.txt`, `webCat1.txt` and `webCat2.txt` contain all the information needed to build the `webCat` - but in textual form.

An extract of the file `webCat1.txt` is shown in Table 10.18. The files `webCat1.txt` and `webCat2.txt` has the same structure as `webCat0.txt` containing pairs of lines with title and associated uri of documentation web-pages:

```
...
Microsoft.FSharp.Collections Namespace (F#)
http://msdn.microsoft.com/en-us/library/ee353413
Microsoft.FSharp.Control Namespace (F#)
http://msdn.microsoft.com/en-us/library/ee340440
Microsoft.FSharp.Core Namespace (F#)
http://msdn.microsoft.com/en-us/library/ee353649
...
```

Table 10.18 *Extract of the file* `webCat1.txt`

```
open System.IO;;
open TextProcessing;;

let addOneRef (e: Map<string,string>) (rd: StreamReader) =
  Map.add (rd.ReadLine()) (rd.ReadLine()) e;;

let addRefsInFile (e: Map<string,string>) path =
  fileXfold addOneRef e path;;

[<EntryPoint>]
let main (args: string[]) =
  let webCat = Array.fold
                 addRefsInFile
                 Map.empty
                 args
  saveValue webCat "webCat.bin"
  0;;
```

Table 10.19 *The file* `MakeWebCat.fsx`

Generating `webCat.bin`

This is done using the `MakeWebCat` program shown in Table 10.19. A free-standing version of this program is called as follows in a command window:

```
MakeWebCat webCat0.txt webCat1.txt webCat2.txt
```

The program inputs the title-uri pairs in each of the specified webCat input files and collects the corresponding webCat entries in the `webCat` map. This map is then saved in the file `webCat.bin` using the `saveValue` function in the books `TextProcessing` library.

Using the file `MakeWebCat.fsx` in an interactive environment one should make the following call of the `main` function:

```
main [| "webCat0.txt";  "webCat1.txt";  "webCat2.txt" |];;
val it : int = 0
```

Generating the `index.html` *file*

The stage is now set for calling the `IndexGen` program described in Section 10.8. A free-standing `IndexGen` program is called:

```
IndexGen
```

from a command window, assuming that the `webCat.bin` and `keywords.txt` are placed in the same directory as the program. The program restores the `webCat` map from the binary file `webCat.bin` and reads the text file `keywords.txt`. The data are captured as described in Section 10.8 and each keyword is combined with the uri found by searching the corresponding title in the `webCat` map. Upon that the `index.html` file is produced by the `webOut` function.

Using the file `IndexGen` in an interactive environment one makes the following call of its `main` function

```
main [||];;
val it : int = 0
```

Summary

In this chapter we have studied a part of the text processing facilities available in F# and the .NET library, including

- regular expressions,
- textual input and output from and to files,
- save and retrieval of F# values on and from files,
- culture-dependant ordering of strings,
- retrieval of web-information, and
- processing XML files.

These text-processing facilities are generally applicable, and we have illustrated their expressive power in the construction of a program that can generate a web-page containing a keyword index of the F# and .NET library documentation.

Exercises

10.1 The term "word" is used in this exercise to denote a string not containing blank characters. The blank characters of a string do hence divide the string into words. Make a program `WordCount` that is called with two parameters:

`WordCount` *inputFile outputFile*

The program should read the input file and produce an output file where each line contains a word found in the input file together with the number of occurrences, for example, "`peter 3`". The words should appears in alphabetic order and the program should not distinguish between small and capital letters.

10.2 The HTML elements `<pre> ... </pre>` encloses a *pre-formatted* part of the web-page. This part is displayed exactly as written, including spaces and line breaks, but each line should, of course, be encoded in HTML encoding. Parts of this text can be copied using copy-paste when the page is displayed using a web-browser. Make a program with program call

`examplePage` *fileName*`.txt`

that inputs the contents of the text file *fileName*`.txt` and produces a web-page *fileName*`.html` containing the contents of this file as preformatted text.

10.3 This exercise is a continuation of Exercise 10.1.

1. We do not consider the hyphen character "–" a proper character in a word. Make a function to capture the list of words in a string while removing any hyphen character.
2. Make a function of type `string -> (string list)*(string option)` removing hyphen characters like the previous one, but treating the last word in the line in a special way: we get the result

   `([`$word_0$`; ...;`$word_{n-1}$`],None)`

   if the last word in the string does not end with a hyphen character, and

   `([`$word_0$`; ...;`$word_{n-2}$`], Some` $word_{n-1}$`)`

   if the last word terminates with a hyphen character.

Make a new version of the `WordCount` program in Exercise 10.1 that in general ignores hyphen characters but handles words that are divided from a text line to the next by means of a hyphen character.

10.4 A position on Earth is given by geographic longitude and latitude written in the form:

`14°27′35.03" E 55°13′47" N`

containing degrees, minutes (1/60 degrees) and seconds (1/60 minutes) where the letters `E` or `W` denote positive or negative sign on a longitude while `N` or `S` denote positive or negative sign on an latitude. The seconds (here: 35.03) may have a decimal point followed by decimals or it may just be an integer. The unit symbols (`°` and `′` and `"`) are assumed to consist of one or several non-digit and non-letter characters. Make a program to capture a position from a string as a value of type `float*float`.

10.5 Make alternative versions of the programs `IndexGen` and `WebCatGen` in the keyword index problem, where `WebCat` and `KeyWdSet` are represented using the imperative collections `Dictionary` and `SortedSet` (see Section 8.11). The programs should use `iter` functions to build these imperative collections. The files `keywords.txt`, `webCat0.txt`, `webCat1.txt` and `webCat2.txt` can be found on the web-page of the book.