Sequencing, and I/O

Björn Lisper
School of Innovation, Design, and Engineering
Mälardalen University

bjorn.lisper@mdh.se
http://www.idt.mdh.se/~blr/

Sequencing in F#

We said functional programming is about calculating expressions

Simple way of interacting: type an expression, obtain the calculated result

But sometimes, side effects are needed

An example: I/O

Therefore, F# provides a simple way to evaluate expressions in *sequence*:

```
e1 ; e2
```

First evaluate `e1`, then `e2`. Return the value of `e2`

Type of `e1 ; e2 =` type of `e2`

With `#light` syntax, sequencing can be done by placing the expressions on different lines instead:

```
e1
e2
```

E.g.

```
"Nisse"
35 + 56
```

Returns `91`, with type `int`

Side Effects

What's the point of this?

It seems unnecessary to evaluate `e1` in `e1 ; e2`

But F# is not a pure functional language. Evaluating expressions can have *side effects*

The order of side effect matters

A Simple Print Function

F# has a function `printf`

Very similar to `printf` in other languages

It takes a format string and a number of additional arguments

```
printf argument-string arguments
```

It prints the values of the arguments according to the formatting string

```
printf "n: %d, x: %f\n" 17 3.0 → n: 17, x: 3.000000
```

Only this side effect is of interest, returns nothing useful

Simple Sequencing Example with printf

```
printf "n: %d, x: %f" 17 3.0
printf " skonummer %d\n" 43
```

will yield the printout

```
n: 17, x: 3.000000 skonummer 43
```

What printf Returns

F# has a data type `unit`

It has a single value "`()`"

Functions like `printf`, which only are executed for their side effect, return `()`

This indicates that they don't return anything useful

Corresponds to the `void` data type in other languages

Sequencing with Return of Useful Values

The ability to return values from sequenced expression can be useful

For instance, flexible ways of doing debug printouts

An example: a function `traceint` that can be used to trace the values of integer expressions in functions:

```
let traceint n = printf "%d " n; n
```

A factorial function that prints the argument that its called with for each call:

```
let rec fac n = if n = 0 then 1 else n * fac (traceint (n-1))
```

Actually, functional programming is very good for testing purposes. Easy to script test suites directly in the language, and instrument the code with debug printouts

A Subtle Thing with Side Effects in F#

Side effects occur when the code is executed

Sometimes, this happens already when a value is declared:

```
let nuff = printf "xxx\n" ; 2 + 2
```

Here, `nuff` will be evaluated directly into `4`

`xxx` will be printed when the expression in the declaration is evaluated

When `nuff` is used in the program `4` will be returned, but no printout!

This behavior can be avoided by turning the declared entity into a function

When a function is called, its body is evaluated over again, with the actual arguments

Therefore, the side effect occurs every time the function is called

```
let nuff n = printf "xxx\n" ; 2 + 2
nuff : 'a -> int
```

`xxx` will now be printed every time `nuff` is called

Simple File I/O

F# has a namespace `System.IO`, which contains means for communicating with the surrounding world

In particular to write and read *files*:

```
open System.IO // Name spaces can be opened just as modules

File.WriteAllText("test.txt", "Allan tar kakan\n och makan")

let s = File.ReadAllText("test.txt")
```

First writes a string to the file `test.txt`, then reads back the string and binds `s` to it

So `File.WriteAllText` has the *side effect* of creating a file, and writing a string to that file

Note the syntax:

```
File.WriteAllText("test.txt", "Allan tar kakan\n och makan")
```

`File.WriteAllText` does not have the usual function syntax of F#

It uses syntax from the *object-oriented* part of F#

`File` can be seen as an object representing the whole file system

`File.WriteAllText` is a *method* affecting the state of the file system

(Methods are called *members* in F#)

Member calls use dot notation, and parentheses around arguments

We can of course define a wrapper function if we prefer functional syntax:

```
let file_write_alltext file string =
  File.WriteAllText(file, string)
```

In general, the object-oriented part of F# comes into play when interfacing with the .NET environment

More on F# and object-orientation later

Some More File I/O

`File.WriteAllText` writes a string to a whole file in one go, and `File.ReadAllText` reads the whole content of a file into a string

Not efficient for large files. For such files, better to process them the conventional way:

- Open the file
- Read (or write) line by line
- Close the file

Some Simple .NET Stream I/O in F#

F# has support for this. Objects of type `StreamReader` and `StreamWriter` represent files open for read and write access, respectively

```
open System.IO
let myfile = File.CreateText("arne.txt")
  // create a new file "arne.txt", open it for write access,
  // create a StreamWriter object representing it, and bind
  // myfile to that object
myfile.WriteLine("Hello World") // write a line to the file
myfile.WriteLine("Hello World 2") // write a second line
myfile.Close() // close the file
```

Think of `myfile` as a *handle* to the file

Some Types

```
File.CreateText("arne.txt") : StreamWriter
```

```
myfile.WriteLine("Hello World") : unit
```

```
myfile.Close() : unit
```

`myfile.WriteLine(...)` and `myfile.Close()` don't return anything sensible, thus they have type `unit`

But `File.CreateText("arne.txt")` returns a `StreamWriter` object (file handle) and thus has type `StreamWriter`

A StreamReader Example

```
open System.IO
let myfile = File.OpenText("arne.txt")
  // open the file "arne.txt" for read access,
  // create a StreamReader object representing it, and bind
  // myfile to that object
let s1 = myfile.ReadLine() // read first line from the file
let s2 = myfile.Readline() // read second line
let lines = (s1,s2) // tuple with the two first lines
myfile.Close() // close the file

File.OpenText("arne.txt") : StreamReader
```

An Example: Turning Whitespace into Single Space

Remember `string2words`?

We can use it to “tidy” text files by turning all whitespace between words into a single space

Let’s use the version that works on strings:

```
string2words : int * string -> string list
```

Read text from file `in.txt`, write “tidied” text to `out.txt`

For simplicity, we will use `File.WriteAllText` and `File.ReadAllText`

Solution on next slides . . .

`File.WriteAllText` writes a string to the file, not a list of strings

We need a function that converts a list of strings (words) into a single string, with a single space in-between each word

Any idea how to define it?

(Solution on next slide)

Converting List of Words to String

```
let rec words2string ws =
  match ws with
  | []         -> ""
  | w :: rest -> w + " " + words2string rest

words2string : string list -> string
```

This solution has a deficiency: it puts a space after the last word

Exercise: declare an improved version of `words2string` which avoids this!

A Wrapper for string2words

`string2words` has an extra position argument (`int`)

This argument is used to keep track of the current position in the string

For first call to `string2words`, it is zero

A wrapper function that calls `string2words` with first argument $= 0$:

```
let string_2_words s = string2words (0,s)
```

```
string_2_words : string -> string list
```

(We could have avoided the declaration with the use of nameless functions. More on them later)

Putting it all Together

A way to do it:

1. Read contents of file `in.txt` into string
2. apply `string_2_words` to string
3. apply `words2string` to result
4. Write result of this to `out.txt`

A solution on next slide . . .

```
let s1 = File.ReadAllText("in.txt")
let w  = string_2_words s1
let s2 = words2string w
File.WriteAllText("out.txt",s2)
```

Note the separation of purely functional parts (`string_2_words`, `words2string`) and parts with side effects (`File.WriteAllText`). It is usually good practice to write software this way

Same Solution, Different Style

We can get rid of the intermediate variables `s1`, `w`, `s2` by directly applying functions to result of other functions:

```
File.WriteAllText("out.txt",
                  words2string
                    (string_2_words File.ReadAllText("in.txt")))
```

No intermediates, but maybe not so easy to read

Can we use a different syntax to make this easier?

The “Forward Pipe” Operator

F# has a “forward pipe” binary operator: `|>`

Definition:

```
let (|>) x f = f x
```

It’s just another way to write function application! What’s the point with this?

We can replace `words2string ...` with `... |> words2string`

Similar to unix pipes: “`|`”

Typically used to “pipe” several functions with one argument

A Forward Pipe Solution

```
let s = File.ReadAllText("in.txt")
        |> string_2_words
        |> words2string
in File.WriteAllText("out.txt",s)
```

Presumably easier to read and understand

Think of `|>` as “pass data from left to right”

This is typical functional programming style!

An Example with Recursion

Let's define a function that writes a number of lines to a file, each differing only in line numbering, like this:

```
Line no. 1
Line no. 2
Line no. 3
...
```

The number of lines shall be a parameter, as the file name

(See next slides for solution)

Solution, Overview

We will split the solution into two parts;

- One part that reads the file name and number of lines, opens the file, calls a “print function” that writes the lines, and closes the file
- One part that is the “print function”. This will be the recursive part

We will use a `StreamWriter` object to write line by line

Solution, Part 1

Assume the print function is `printline file n`, where `file` is the `StreamWriter` object and `n` is the number of lines to write;

```
let writelines filename n =
  let file = File.CreateText(filename)
  printline file n
  file.Close()
```

Solution, Part 2

To create strings we can use `sprintf`, a variation of `printf` that writes to a string instead of the console

`printline` will be a wrapper that calls a local, recursive function:

```
let printline file n =
  let rec printlocal k =
    if k < n then file.WriteLine(sprintf "Line no. %d" k)
                  printlocal (k+1)
             else ()
in printlocal 1
```

Functions used only for side effect, returns `()` (type `unit`)

Compare this to a loop in an “ordinary” language!