KSFUPRO1K1U – Functional Programming

Lecture 1: Introduction and Getting Started

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.

WELCOME to
KSFUPRO1K1U – Functional Programming

Teacher: Niels Hallenberg, nh@itu.dk
Peter Sestoft, sestoft@itu.dk
IT University of Copenhagen

Teaching assistant: Jonas Christiansen Røygaard (TA), jocr@itu.dk
Jonas Ishøj Nielsen (TA), join@itu.dk
IT University of Copenhagen

Homepage: `https://learnit.itu.dk/course/view.php?id=3018548`

Practical Matters

- Textbook: Functional Programming using F#
  by Michael R. Hansen and Hans Rischel.

  ISBN: 9781107019027

  Book homepage:
  `http://www2.imm.dtu.dk/~mire/FSharpBook/`

  Published on Cambridge University Press, May 2013
- F# is an open-source functional language integrated in the Visual Studio development platform and with access to all features in the .NET program library.
- You can use F# on all major platforms: Windows, Linux and Mac.
- Lectures, Tuesday 08.00 – 10.00 in Aud 3.
- Classes, Tuesday in room 4A54, 5A60 from 10.00 – 12.00.

Exam and Mandatory Assignments

Exam information is kept updated here:
`https://learnit.itu.dk/mod/wiki/create.php?wid=82&group&uid=12544&title=Exam`.

- **Date and Time:** May 16, 2019
- Exam Syllabus: Functional Programming using F#, Michael R. Hansen and Hans Rischel, ISBN 9781107684065, Chapter 1 - 13.
- **More to come** . . .

You must pass the Mandatory assignments in order to attend exam.
See course homepage in LearnIt for the details.

- Mandatory assignments does improve exam results!
- Main purpose of feedback on assignments is learning - not precise counting of points.
- If you are not satisfied with the grading, then just handin again with your corrections
- Remember the cut-off date - it is very important.

Intended Learning Outcome

After this learning activity you should be able to:

1. apply and reflect on theories for modelling, analyzing and constructing functional declarative programs.
2. apply and reflect on the concepts behind functional programming compared to imperative and object oriented programming.
3. construct programs in F# and explain the basic principles behind functional programming using F#.
4. describe and explain solutions to problems in the context of functional programming.
5. apply core concepts of functional programming.
6. reason about the complexity of functional programs.

Course Content

The subject of the course is functional, declarative programming in general and F# in particular. This includes the following themes:

- Functional Programming Paradigme:
  - first class functions
  - higher-order functions
  - type inference and polymorphism
  - recursion and tail-recursion
  - algebraic data types
  - strict and lazy evaluation
- Memory Management:
  - garbage collection
  - reference types
  - mutable versus immutable data
- Parallel Programming:
  - divide and conquer

Outline today

1. Introduction
   - Historic perspective
   - Imperative models
   - Object-oriented models
   - Declarative models
   - Functional programming background
2. Getting started with F#
   - The interactive environment
   - Declarations
   - Recursion
   - Types
3. Brief introduction to lists
   - Values
   - Types
   - Recursion on lists

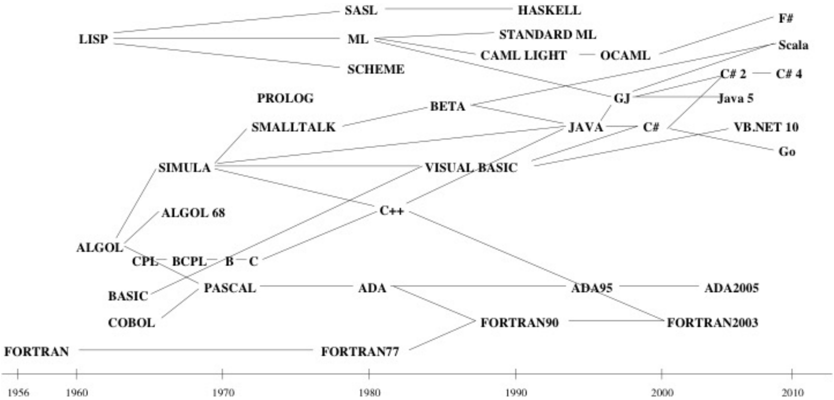
Programming Languages in a historic perspective

Url: `http://en.wikipedia.org/wiki/History_of_programming_languages`

- Lisp (concept, 1956, implementation 1959)
- ML (1973)
- Scheme (1975)
- Standard ML (1984)
- Common LISP (1984)
- Miranda (1986)
- Erlang (1987)
- Haskell (1990)
- Java (1995)
- C# (2000)
- F# (2005)
- Scala (2004, first release)

Url: `http://en.wikipedia.org/wiki/Timeline_of_programming_languages`

The Diagram


SASL
HASKELL
F#
LISP
ML
STANDARD ML
CAML LIGHT
OCAML
Scala
SCHEME
C# 2
C# 4
PROLOG
BETA
GJ
Java 5
SMALLTALK
JAVA
C#
VB.NET 10
Go
SIMULA
VISUAL BASIC
C++
ALGOL 68
ALGOL
CPL
BCPL
B
C
PASCAL
ADA
ADA95
ADA2005
BASIC
COBOL
FORTRAN90
FORTRAN2003
FORTRAN
FORTRAN77
1956
1960
1970
1980
1990
2000
2010

Imperative models

- Imperative models of computations are expressed in terms of states and sequences of state-changing operations

Example:

```
i := 0;
s := 0;
while i < length(A)
  do s := s+A[i];
     i := i+1
  od
```

An imperative model describes *how* a solution is obtained

Object-oriented models

- An object is characterized by a state and an interface specifying a collection of state-changing operations.
- Object-oriented models of computations are expressed in terms of a collection of objects which exchange messages by using interface operations.

Object-oriented models add structure to imperative models

An object-oriented model describes *how* a solution is obtained

Declarative models

In declarative models focus is on *what* a solution is.

- Functional programming
  - A program is expressed as a mathematical function

$$f : A \to B$$

  and evaluations of function applications guide computations.

Some advantages

- fast prototyping based on abstract concepts
- easier reasoning about the smaller features (functions) to build larger features (application of functions).
- execute in parallel on multi-core platforms

F# is as efficient as C#

Some functional programming background

In functional programming, the model of computation is the application of functions to arguments. no side-effects

- Introduction of λ-calculus around 1930 by Church and Kleene when investigating function definition, function application, recursion and computable functions. For example, $f(x) = x + 2$ is represented by $\lambda x.x + 2$.
- Introduction of the type-less functional-like programming language LISP was developed by McCarthy in the late 1950s.
- Introduction of functional languages with a strong type system like ML (by Milner) and Miranda (by Turner) in the 1970s.

Some background of the “SML-family”

- Standard Meta Language (SML) was originally designed for theorem proving

    Logic for Computable Functions (Edinburgh LCF)

    Gordon, Milner, Wadsworth (1977)

- High quality compilers, e.g. Standard ML of New Jersey and Moscow ML, based on a *formal semantics*

    Milner, Tofte, Harper, MacQueen 1990 & 1997

- SML-like systems (SML, OCAML, F#, . . .) have now applications far away from its origins

    Compilers, Artificial Intelligence, Web-applications, Financial sector, . . .

- F# is integrated in the .net environment

- Declarative aspects are sneaking into more ”main stream languages”

- Often used to teach high-level programming concepts

A major goal

Teach abstraction (not a concrete programming language)

- Modelling
- Design
- Programming

Why?

More complex problems can be solved in an succinct, elegant and understandable manner

How?

Solving a broad class of problems showing the applicability of the theory, concepts, techniques and tools.

Functional programming techniques once mastered are useful for the design of programs in other programming paradigms as well.

Outline today

1. Introduction
   - Historic perspective
   - Imperative models
   - Object-oriented models
   - Declarative models
   - Functional programming background
2. Getting started with F#
   - The interactive environment
   - Declarations
   - Recursion
   - Types
3. Brief introduction to lists
   - Values
   - Types
   - Recursion on lists

F# supports

- Functions as first class citizens
- Structured values like lists, trees, . . .
- Strong and flexible type discipline, including

  type inference and polymorphism
- Imperative and object-oriented programming

  assignments, loops, arrays, objects, Input/Output, etc.

Programming as a modelling discipline

- High-level programming, declarative programming
- Fast prototyping

Overview of Getting Started

Main functional ingredients of F#:

- The interactive environment (Visual Studio, Visual Code, Emacs, ...)
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

GOAL: By the end of this first part you have constructed succinct, elegant and understandable F# programs, e.g. for

- $\mathrm{sum}(m, n) = \sum_{i=m}^{n} i$
- Fibonacci numbers ($F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$)
- Binomial coefficients $\binom{n}{k}$

The Interactive Environment

```
2*3 + 4;;                  ⇐ Input to the F# system
val it : int = 10          ⇐ Answer from the F# system
```

- The *keyword* `val` indicates a value is computed
- The *integer* 10 is the computed value
- `int` is the *type* of the computed value
- The *identifier* `it` names the (last) computed value

The notion *binding* explains which entities are named by identifiers.

`it` $\mapsto$ 10 reads: “`it` is bound to 10”

Value Declarations

A value declaration has the form: `let` *identifier* $=$ *expression*

```
let price = 25 * 5;;          ⇐ A declaration as input
val price : int = 125         ⇐ Answer from the F# system
```

The effect of a declaration is a binding: `price` $\mapsto$ 125

Bound identifiers can be used in expressions and declarations, e.g.

```
let newPrice = 2*price;;
val newPrice : int = 250

newPrice > 500;;
val it : bool = false
```

A collection of bindings

$$\begin{bmatrix} \texttt{price} & \mapsto & 125 \\ \texttt{newPrice} & \mapsto & 250 \\ \texttt{it} & \mapsto & \text{false} \end{bmatrix}$$

is called an environment

Function Declarations 1: `let` $f\,x$ = e

Declaration of the circle area function:

```
let circleArea r = System.Math.PI * r * r;;
```

- `System.Math` is a program library
- `PI` is an identifier (with type `float`) for π in `System.Math`

The type is automatically inferred in the answer:

```
val circleArea : float -> float
```

Applications of the function:

```
circleArea 1.0;; (* this is a comment *)
val it : float = 3.141592654

circleArea(3.2);; // A comment: optional brackets
val it : float = 32.16990877
```

Anonymous functions: by example (1)

An anonymous function computing the number of days in a month:

```
function
| 1 -> 31 // January
| 2 -> 28 // February // not a leap year
| 3 -> 31 // March
| 4 -> 30 // April
| 5 -> 31 // May
| 6 -> 30 // June
| 7 -> 31 // July
| 8 -> 31 // August
| 9 -> 30 // September
| 10 -> 31 // October
| 11 -> 30 // November
| 12 -> 31;;// December
... warning ... Incomplete pattern matches ...
val it : int -> int = <fun:clo@17-2>

it 2;;
val it : int = 28
```

A functional expression with a pattern for every month

Anonymous functions: by example (2)

One *wildcard pattern* _ can cover many similar cases:

```
function
 | 2  -> 28  // February
 | 4  -> 30  // April
 | 6  -> 30  // June
 | 9  -> 30  // September
 | 11 -> 30  // November
 |  _  -> 31;;// All other months
```

An even more succinct definition can be given using an *or*-pattern:

```
function
 | 2          -> 28  // February
 | 4|6|9|11 -> 30  // April, June, September, November
 | _          -> 31  // All other months
;;
```

Anonymous functions: by example (2)

One *wildcard pattern* _ can cover many similar cases:

```
function
 | 2  -> 28  // February
 | 4  -> 30  // April
 | 6  -> 30  // June
 | 9  -> 30  // September
 | 11 -> 30  // November
 |  _  -> 31;;// All other months
```

An even more succinct definition can be given using an *or*-pattern:

```
function
 | 2         -> 28  // February
 | 4|6|9|11 -> 30  // April, June, September, November
 | _         -> 31  // All other months
;;
```

Recursion. Example $n! = 1 \cdot 2 \cdot \ldots \cdot n$, $n \geq 0$

Mathematical definition: recursion formula

$$
\begin{aligned}
0! &= 1 && (i) \\
n! &= n \cdot (n-1)!, \quad \text{for } n > 0 && (ii)
\end{aligned}
$$

Computation:

$$
\begin{aligned}
& 3! \\
= \; & 3 \cdot (3-1)! && (ii) \\
= \; & 3 \cdot 2 \cdot (2-1)! && (ii) \\
= \; & 3 \cdot 2 \cdot 1 \cdot (1-1)! && (ii) \\
= \; & 3 \cdot 2 \cdot 1 \cdot 1 && (i) \\
= \; & 6
\end{aligned}
$$

Recursive declaration. Example $n!$

Function declaration:

```
let rec fact = function
  | 0 -> 1                   (* i *)
  | n -> n * fact(n-1);;     (* ii *)
val fact : int -> int
```

Evaluation:

$$
\begin{array}{lllll}
 & \texttt{fact}(3) & & \\
\rightsquigarrow & 3 * \texttt{fact}(3-1) & (ii) & [\texttt{n} \mapsto 3] \\
\rightsquigarrow & 3 * 2 * \texttt{fact}(2-1) & (ii) & [\texttt{n} \mapsto 2] \\
\rightsquigarrow & 3 * 2 * 1 * \texttt{fact}(1-1) & (ii) & [\texttt{n} \mapsto 1] \\
\rightsquigarrow & 3 * 2 * 1 * 1 & (i) & [\texttt{n} \mapsto 0] \\
\rightsquigarrow & 6 & &
\end{array}
$$

$e_1 \rightsquigarrow e_2$ reads: e_1 evaluates to e_2

Recursion. Example $x^n = x \cdot \ldots \cdot x$, n occurrences of x

Mathematical definition: recursion formula

$$
\begin{aligned}
x^0 &= 1 && (1)\\
x^n &= x \cdot x^{n-1}, \quad \text{for } n > 0 && (2)
\end{aligned}
$$

Function declaration:

```
let rec power = function
  | (_,0) -> 1.0                    (* 1 *)
  | (x,n) -> x * power(x,n-1)   (* 2 *)
```

Patterns:

$(\_, 0)$ matches any pair of the form $(x, 0)$.
The wildcard pattern _ matches any value.

(x, n) matches any pair (u, i) yielding the bindings

$$\mathtt{x} \mapsto u, \mathtt{n} \mapsto i$$

Evaluation. Example: `power(4.0, 2)`

Function declaration:

```
let rec power = function
  | (_,0) -> 1.0                 (* 1 *)
  | (x,n) -> x * power(x,n-1)   (* 2 *)
```

Evaluation:

$$
\begin{array}{lll}
 & \mathtt{power}(4.0, 2) & \\
\rightsquigarrow & 4.0 * \mathtt{power}(4.0, 2-1) & \text{Clause 2, } [\mathtt{x} \mapsto 4.0, \mathtt{n} \mapsto 2] \\
\rightsquigarrow & 4.0 * \mathtt{power}(4.0, 1) & \\
\rightsquigarrow & 4.0 * (4.0 * \mathtt{power}(4.0, 1-1)) & \text{Clause 2, } [\mathtt{x} \mapsto 4.0, \mathtt{n} \mapsto 1] \\
\rightsquigarrow & 4.0 * (4.0 * \mathtt{power}(4.0, 0)) & \\
\rightsquigarrow & 4.0 * (4.0 * 1) & \text{Clause 1} \\
\rightsquigarrow & 16.0 &
\end{array}
$$

If-then-else expressions

Form:

`if` b `then` e_1 `else` e_2

Evaluation rules:

$$\texttt{if true then } e_1 \texttt{ else } e_2 \rightsquigarrow e_1$$
$$\texttt{if false then } e_1 \texttt{ else } e_2 \rightsquigarrow e_2$$

Alternative declarations:

```
let rec fact n        =   if n=0 then 1
                          else n * fact(n-1);;

let rec power(x,n)    =   if n=0 then 1.0
                          else x * power(x,n-1);;
```

Use of patterns often give more understandable programs

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `&&` e_2 “conjunction $e_1 \wedge e_2$”
e_1 `||` e_2 “disjunction $e_1 \vee e_2$”

— are lazily evaluated (short circuit eval.), e.g.

```
1<2 || 5/0 = 1
⇝ true
```

Precedence: `&&` has higher than `||`

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`String.length`	`string -> int`	length of string
`+`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`string`	`obj -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"+"de";
> val it = "abcde": string
```

```
- String.length("abc"^"def");
> val it = 6 : int

- string(6+18);
> val it = "24": string
```

Types — every expression has a type $e : \tau$

Basic types:

	type name	example of values
Integers	`int`	~27, 0, 15, 21000
Floats	`float`	~27.3, 0.0, 48.21
Booleans	`bool`	true, false

Pairs:

If $e_1 : \tau_1$ and $e_2 : \tau_2$

then $(e_1, e_2) : \tau_1 * \tau_2$ pair (tuple) type constructor

Functions:

if $f : \tau_1$ -> τ_2 and $a : \tau_1$ function type constructor

then $f(a) : \tau_2$

Examples:

```
(4.0, 2):  float*int
power:  float*int -> float
power(4.0, 2):  float
```

$\star$ has higher precedence than ->

Type inference: `power`

```
let rec power = function
  | (_,0) -> 1.0                 (* 1 *)
  | (x,n) -> x * power(x,n-1) (* 2 *)
```

- The type of the function must have the form: τ_1 `*` τ_2 `->` τ_3, because argument is a pair.
- τ_3 = `float` because `1.0:float` (Clause 1, function value.)
- τ_2 = `int` because `0:int`.
- `x*power(x,n-1):float`, because τ_3 = `float`.
- multiplication can have

  `int*int -> int` or `float*float -> float`

  as types, but no "mixture" of `int` and `float`
- Therefore `x:float` and τ_1=`float`.

The F# system determines the type `float*int -> float`

Summary

- The interactive environment
- Values, expressions, types, patterns
- Declarations of values and recursive functions
- Binding, environment and evaluation
- Type inference

Outline

Overview

- Lists: values and constructors
- Recursions following the structure of lists

The purpose of this lecture is to give you an (as short as possible) introduction to lists, so that you can solve a problem which can illustrate some of F#'s high-level features.

This part is *not* intended as a comprehensive presentation on lists, and we will return to the topic again later.

Lists

A list is a finite sequence of elements having the same type:

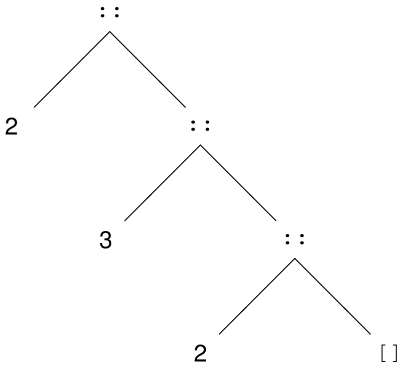
$[v_1; \ldots; v_n]$ ([ ] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]

[(1,true); (3,true)];;
val it : (int * bool) list = [(1, true); (3, true)]

[[]; [1]; [1;2]];;
val it : int list list = [[]; [1]; [1; 2]]
```

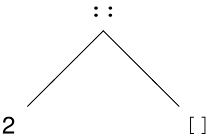
Trees for lists

A non-empty list $[x_1, x_2, \ldots, x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2, \ldots, x_n]$



Graph for `[2,3,2]`



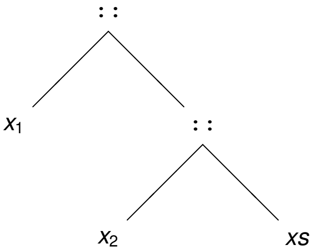
Graph for `[2]`

List constructors: [ ] and ::

Lists are generated as follows:

- the empty list is a list, designated [ ]
- if x is an element and xs is a list, then so is $x :: xs$ (type consistency)

:: associate to the right, i.e. $x_1{::}x_2{::}xs$ means $x_1{::}(x_2{::}xs)$



Graph for $x_1 :: x_2 :: xs$

Recursion on lists – a simple example

$$\texttt{suml}\ [x_1, x_2, \ldots, x_n] = \sum_{i=1}^{n} x_i = x_1 + x_2 + \cdots + x_n = x_1 + \sum_{i=2}^{n} x_i$$

Constructors are used in list patterns

```
let rec suml = function
  | []     -> 0
  | x::xs  -> x + suml xs;;
> val suml : int list -> int
```

	`suml [1;2]`	
$\rightsquigarrow$	`1 + suml [2]`	($\texttt{x} \mapsto 1$ and $\texttt{xs} \mapsto$ `[2]`)
$\rightsquigarrow$	`1 + (2 + suml [])`	($\texttt{x} \mapsto 2$ and $\texttt{xs} \mapsto$ `[]`)
$\rightsquigarrow$	`1 + (2 + 0)`	(the pattern `[]` matches the value `[]`)
$\rightsquigarrow$	`1 + 2`	
$\rightsquigarrow$	`3`	

Recursion follows the structure of lists

Infix functions

It is possible to declare infix functions in F#, i.e. the function symbol is between the arguments.

The *prefix function* on lists is declared as follows:

```
let rec (<=.) xs ys =
   match (xs,ys) with
   | ([],_) -> true
   | (_,[]) -> false
   | (x::xs',y::ys') -> x<=y &&  xs' <=. ys';;

 [1;2;3] <=. [1;2];;
val it : bool = false
```

- The special way of declaring the function `(<=.) xs ys` makes `<=.` an infix operator
- The `match (xs,ys)` construct allows for branching out on patterns for `(xs,ys)`

Suitable use of infix functions can increase readability significantly