# 02157 Functional Programming

Lecture 3: Programming as a model-based activity

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

# Overview

- Syntax, semantics and pragmatics (briefly)
  - Overview of F#
- Programming as a modelling activity
  - Cash register
  - Map colouring
- Program properties and property-based testing

# Syntax, semantics and pragmatics

# Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.
- The semantics is concerned with the meaning of syntactically correct programs.
- The pragmatics is concerned with practical (adequate) application of language constructs in order to achieve certain objectives.

A specification of F# is found at
`https://fsharp.org/specs/language-spec/`

# Further characteristics for the functional fragment of F#

F# is a statically typed, compiled language language:

- At compile time: A type for every expression in a program is inferred.
  If this is not possible, then an type error is issued at compile time

- Code is only generated by the compiler for well-typed programs.

- At runtime: The generated code contains no type information
  well-typed programs do not go wrong

| Syntax | Static semantics<br>Type inference $e : \tau$ | Semantics |
|---|---|---|
| Types $\tau$<br>Patterns *pat*<br>Expressions $e$<br>Declarations $d$ | <br><br>Types every piece of an expression<br>Types every piece of a declaration | Value $v$<br>Binding $id \mapsto v$<br>Environment<br>$e_1 \rightsquigarrow e_2$ |
| indentation<br>sensitive | | |

Pragmatics: ?

- type and function names are descriptive
- types start with a capital letter
- variables names are short and consistently used
- function types are stated in comments
- a program is composed by small, well-understood pieces
- adequate use of language constructs
- ...
- use common computer-science sense

# Syntactical constructs in F#

- Constants: 0, 1.1, true, ...
- Patterns:
  $x \quad \_ \quad (p_1, \ldots, p_n) \quad p_1 :: p_2 \quad p_1 | p_2 \quad p \texttt{ when } e \quad p \texttt{ as } x \quad p : t \ldots$
- Expressions:
  $x \quad (e_1, \ldots, e_n) \quad e_1 :: e_2 \quad e_1 e_2 \quad e_1 \oplus e_2 \quad \texttt{let } p_1 = e_1 \texttt{ in } e_2 \quad e : t$

  $\texttt{match } e \texttt{ with } \textit{clauses} \quad \texttt{fun } p_1 \cdots p_n \texttt{->} e \quad \texttt{function } \textit{clauses} \quad \ldots$
- Declarations $\texttt{let } f\ p_1 \ldots p_n = e \quad \texttt{let rec } f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  $\texttt{int} \quad \texttt{bool} \quad \texttt{string} \quad 'a \quad t_1 * t_2 * \cdots * t_n \quad t \texttt{ list} \quad t_1 \texttt{->} t_2 \ldots$

where the construct *clauses* has the form:

$\texttt{|}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n$

In addition to that

- precedence and associativity rules, parenthesis around *p* and *e* and type correctness

# Semantics of a function

Consider a declaration of `f` in an environment $env = [a \mapsto 4, b \mapsto true]$:

```
let f x = x+a
```

The resulting environment is:

$$[a \mapsto 4, b \mapsto true, f \mapsto cl_f]$$

where the *value of f* is a closure $cl_f$ =

$$([x], x + a, [a \mapsto 4])$$

consisting of

- the argument list [x]
- the body of f x+a
- the environment with bindings for the *free* variables $[a \mapsto 4]$

static binding

# Programming as a modelling activity

# Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

# The problem

*An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.*

*The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.*

# A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type ArticleCode = string
type ArticleName = string
type Price       = int
type Register    = (ArticleCode * (ArticleName*Price)) list
type NoPieces    = int
type Item        = NoPieces * ArticleCode
type Purchase    = Item list
type Info        = NoPieces * ArticleName * Price
type Infoseq     = Info list
type Bill        = Infoseq * Price

makeBill: Register -> Purchase -> Bill
```

Is the model adequate?

# Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

# Functional decomposition (1)

Type: findArticle: ArticleCode → Register → ArticleName * Price

```
let rec findArticle ac = function
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                     -> findArticle ac reg
    | _                          ->
          failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

An article description is found as follows:

```
findArticle "a2" reg;;
val it : string * int = ("herring", 4)

findArticle "a5" reg;;
System.Exception: a5 is an unknown article code
   at FSI_0016.findArticle[a] ...
```

Note: failwith is a built-in function that raises an exception

# Functional decomposition (2)

Type: makeBill: Register → Purchase → Bill

```
let rec makeBill reg = function
    | []             -> ([],0)
    | (np,ac)::pur ->
          let (aname,aprice) = findArticle ac reg
          let tprice         = np*aprice
          let (billtl,sumtl) = makeBill reg pur
          ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
    (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int
```

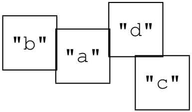
```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

# Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

# Example: Map Coloring.

Color a map so that neighbouring countries get different colors




The types for country and map are “straightforward”:

- `type Country = string`

  Symbols: c, c1, c2, c’; Examples: ”a”, ”b”, . . .

- `type Map=(Country*Country) list`

  Symbols: m; Example: val exMap = [(”a”,”b”); (”c”,”d”); (”d”,”a”)]

How many ways could above map be colored?

# Abstract models for color and coloring

- `type Color = Country list`

  Symbols: col; Example: [”c”; ”a”]

- `type Coloring = Color list`

  Symbols: cols; Example: [[”c”; ”a”]; [”b”; ”d”]]

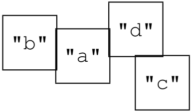
*Be conscious about symbols and examples*

```
colMap: Map -> Coloring
```

| ***Meta symbol: Type*** | | ***Definition*** | ***Sample value*** |
|---|---|---|---|
| `c:` | `Country` | `string` | `"a"` |
| `m:` | `Map` | `(Country*Country) list` | `[("a","b"),("c","d"),("d","a")]` |
| `col:` | `Color` | `Country list` | `["a","c"]` |
| `cols:` | `Coloring` | `Color list` | `[["a","c"],["b","d"]]` |

Figure: A Data model for map coloring problem

# Algorithmic idea




Insert repeatedly countries in a coloring.

| | country | old coloring | new coloring |
|---|---|---|---|
| 1. | "a" | [] | [["a"]] |
| 2. | "b" | [["a"]] | [["a"] ; ["b"]] |
| 3. | "c" | [["a"] ; ["b"]] | [["a";"c"] ; ["b"]] |
| 4. | "d" | [["a";"c"] ; ["b"]] | [["a";"c"] ; ["b";"d"]] |

Figure: Algorithmic idea

# Functional decomposition (I)

To make things easy

Are two countries neighbours?

areNb: Map → Country → Country → bool

```
let areNb m c1 c2 = List.contains (c1,c2) m
                    || List.contains (c2,c1) m;;
```

Can a color be extended?

canBeExtBy: Map → Color → Country → bool

```
let rec canBeExtBy m col c =
  match  col with
  | []        -> true
  | c'::col' -> not (areNb m c' c) && canBeExtBy m col' c;;

canBeExtBy exMap ["c"] "a";;
val it : bool = true

canBeExtBy exMap ["a"; "c"] "b";;
val it : bool = false
```

# Functional composition (I)

Combining functions make things easy

Extend a coloring by a country:

extColoring: Map $\rightarrow$ Coloring $\rightarrow$ Country $\rightarrow$ Coloring

```
Examples:
  extColoring exMap [] "a"        =  [["a"]]
  extColoring exMap [["b"]] "a"   =  [["b"] ; ["a"]]
  extColoring exMap [["c"]] "a"   =  [["a"; "c"]]
```

```
let rec extColoring m cols c =
    match cols with
    | []          -> [[c]]
    | col::cols’  -> if canBeExtBy m col c
                     then (c::col)::cols’
                     else col::extColoring m cols’ c;;
```

*Function types, consistent use of symbols, and examples make program easy to comprehend*

# Functional decomposition (II)

To color a neighbour relation:

- Get a list of countries from the neighbour relation.
- Color these countries

Get a list of countries without duplicates:

```
let addElem x ys = if List.contains x ys then ys else x::ys;;

let rec countries = function
    | []            -> []
    | (c1,c2)::m -> addElem c1 (addElem c2 (countries m));;
```

Color a country list:

```
let rec colCntrs m = function
    | []     -> []
    | c::cs -> extColoring m (colCntrs m cs) c;;
```

# Functional composition (III)

The problem can now be solved by
combining well-understood pieces

Create a coloring from a neighbour relation:

colMap: Map $\rightarrow$ Coloring

```
let colMap m = colCntrs m (countries m);;

colMap exMap;;
val it : string list list = [["c"; "a"]; ["b"; "d"]]
```

# On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.
- Examples may help understanding the problem and its solution.
- Functional paradigm is powerful.

  Problem solving by combination of well-understood pieces

These points are not programming language specific

# Program properties
# and
# property-based testing

# Example: Invariant presevation

An integer list $[x_0; x_1; \ldots; x_{n-1}]$ is ordered if

$$x_0 \leq x_1 \leq \cdots \leq x_{n-1} \quad \text{where } n \geq 0$$

The function:

```
let rec insert y xs =
  match xs with
  | []              -> [y]                (* C1 *)
  | x::_ when y<=x -> y::xs               (* C2 *)
  | x::rest         -> x::insert y rest   (* C3 *)
```

inserting $y$ in an ordered list $xs$ should satisfy the property:

If $xs$ is ordered,
then `insert` $y$ $xs$ is ordered as well.

Informal argument: Assume $xs$ is ordered. There are three cases:

- C1 The singleton list $[y]$ is trivially ordered.
- C2: Since $y$ is smaller than the smallest element $x$ of the ordered list $xs$, the list $y :: xs$ must be ordered.
- C3: Assuming that inserting $y$ in the shorter ordered tail list $rest$ gives an ordered list (Induction hypothesis), we have that `x::insert y rest` is ordered since $x < y$

# Property-based testing

*QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*, Claessen and Hughes, 2000

- Random generation of values of arbitrary types
- Properties are expressed as Boolean-valued functions

```
let rec sort xs = .....
let rec ordered xs = ...

// Test that: for all lists xs: ordered(sort xs)
let sortProp (xs: int list) = ordered(sort xs)

let _ = Check.Quick sortProp
 Ok, passed 100 tests.
```

The tool has been ported to many languages. We look at FsCheck for the .Net platform. Consult

- `https://fscheck.github.io/FsCheck/` and
- TipsTricksPrograms DTU Learn

# Testing for correctness wrt. a reference model (I)

```
#r ".......FsCheck.dll"
open FsCheck

let rec sumA xs acc =
   match xs with
   | []    -> 0
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Quick sumRefProp;;
Falsifiable, after 2 tests (2 shrinks) (StdGen ...... :
Original:
[-2; -1]
Shrunk:
 [1]
```

- uses built-in generators for lists
- tool provides a short counterexample

# Testing for correctness wrt. a reference model (II)

```
let rec sumA xs acc =
   match xs with
   | []     -> acc
   | x::xs -> sumA xs (x+acc);;
```

Correctness property wrt. the built-in function: List.sum:

for all xs: List.sum xs = sumA xs 0

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Quick sumRefProp;;
 Ok, passed 100 tests.
```

- default is 100 random tests
- can be configured

# Testing for correctness wrt. a reference model (III)

Test is exposed using Check.Verbose as follows:

```
let sumRefProp xs = List.sum xs = sumA xs 0;;
let _ = Check.Verbose sumRefProp;;

0:
[-2]
.....
99:
[-1; 0; -1; -1; 2; 1; -1; 0; 0; 5; -1; 1; -1; 0; 0; -1; 2;
1; -1; 1; -1; 0; -1; -1; -1; -1; 1; -1; 1; 1; 1; 0; -2; 1;
1; 0; -1; 0; -1; -1; -2; 2; 0; 1; -1; -1; 1; 1; 0; 0; -1; 0
0]
Ok, passed 100 tests.
```

# Testing involving functions

Consider the `curry` / `uncurry` exercise:

```
let curry f x y = f(x,y);;
let uncurry g (x,y) = g x y;;

// use monomorphic types when properties are tested
let curryProp1 (g:int -> int -> int) x y =
    g x y = (curry (uncurry g)) x y;;

let curryProp1Test = Check.Verbose curryProp1;;
>
0:
<fun:Invoke@2810>
-2
2
.....
99:
...
```

- use monomorphic types (no type variables) when properties are tested
- 100 random functions are generated

# Summary

Property-based testing supports testing at a high level of abstraction

- Focus is on fundamental properties – not on concrete test cases
- You write programs for properties – not concrete test cases
- Properties are tested automatically
- Short counterexamples are found — when properties are falsified

The examples given here are just appetizers.

This is actually all you need to know for Part 5 of the polynomial exercise.