Course introduction
Introduction to F#

FPLI 2023, week 1

Morten Rhiger

Roskilde University

March 6, 2023

Course purpose

What?	Develop a knowledge of programming *languages,* programming-language technologies, and functional programming.
Why?	Become better programmers.
How?	Introduce functional programming.
	Study programming-language constructs and terminology.
	Implement programming languages.

Course content

Part 1. Functional programming in F#

- Expressions, types, values
- Lists, trees, datatypes, pattern matching
- Recursion
- Functions, higher-order functions

Part 2. Implementations of programming languages

- Interpreters
- Compilers (to virtual machines)
- Type checkers

Functional programming

- Expressions having values (instead of statements having side effects)
- Variables & bindings (instead of memory & assignments)
- Functions (instead of procedures or methods)
- Higher-order functions
- Anonymous functions
- Pattern matching (instead of if-statements + selectors)
- Recursion (instead of iteration)
- Static type system (instead of runtime errors)
- Inferred types (instead of explicit annotations)
- Polymorphism (instead of subtyping, overloading)
- Data types (instead of class hierarchies)
- Read-eval-print loop (instead of a `main()` method)

Read-eval-print loop (REPL)

```
Microsoft (R) F# Interactive version 11.4.2.0 for F# 5.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> 1 + 2;;
val it : int = 3
>
```

Type system

- Any legal expression has *a type* $\approx$ set of values.
- F# determines the type of an expression *before* evaluating the expression.
- If no type can be found, the expression is illegal.

Expression		Type
`42`	:	`int`
`4 + 2`	:	`int`
`pow2(-1)`	:	`int`
`"Hello"`	:	`string`
`'H'`	:	`char`
`true`	:	`bool`
`4 < 2`	:	`bool`
`(4 + 2, 4 < 2)`	:	`int * bool`
`pow2`	:	`int -> int`
`pow2 'H'`	:	type error

Tuples

- An N-tuple is a sequence of N values:

```
(2022, 3, 15)                // Today
("Cicero", (-106, 1, 3))  // A Roman, born 106 BC
```

- The types of the element may differ:

Tuples

- An N-tuple is a sequence of N values:

```
(2022, 3, 15)               // Today
("Cicero", (-106, 1, 3))  // A Roman, born 106 BC
```

- The types of the element may differ:
  The type of (e_1, $\cdots$, e_N) is $t_1 * \cdots * t_N$ if e_i has type t_i.
- Tuples can be nested.
- We examine tuples by *pattern matching*.

Tuples are values (1)

Functions can take tuples as input:

```
let year (y, m, d)     = y     // Pattern match
let birthday (name, d) = d     // Pattern match
let age p = 2022 - year (birthday p)

age ("Cicero", (-106, 1, 3))   // 2128 (!)
```

Tuples are values (2)

Functions can return tuples as output:

```
let nextYear (y, m, d) = (y + 1, m, d)
```

Tuples are values (2)

Functions can return tuples as output:

```
let nextYear (y, m, d) = (y + 1, m, d)

let today = (2022, 3, 14)
nextYear today                  // (2023, 3, 14)
```

Example

$$f(x) = ax^2 + bx + c$$

(1) Compute?

(2) Find zeros?

Datatypes; Enumerations

```
type weekday = | MONDAY    | TUESDAY   | WEDNESDAY   | THURSDAY
               | FRIDAY    | SATURDAY  | SUNDAY
```

Datatypes; Enumerations

```
type weekday = | MONDAY    | TUESDAY   | WEDNESDAY   | THURSDAY
               | FRIDAY    | SATURDAY  | SUNDAY
```

defines

- one new type `weekday` and
- seven *constructors*, `MONDAY`, $\cdots$, `SUNDAY`, of that type.

Datatypes; Enumerations

- A datatype is a *new type,* distinct from all other types:

$$\begin{aligned} \texttt{weekday} &\neq \texttt{int}, \\ \texttt{weekday} &\neq \texttt{string} \\ &\vdots \end{aligned}$$

- `weekday` is an `enumeration:` It has just seven values.

  Compare with type `string`:

  `"MOONDAY"` `"WED"` `"tuesday"` `"someday"` `"foo"` `"4"` `""`

Datatypes

- Functions on datatypes are typically implemented using pattern matching:

```
let isWeekend = function
                  | SATURDAY -> true
                  | SUNDAY   -> true
                  | _        -> false
```

Datatypes; Constructors carrying values

```
type intOption = | NONE
                 | SOME of int
```

Datatypes; Constructors carrying values

```
type intOption = | NONE
                 | SOME of int
```

defines

- one new type `intOption`,
- one constructor `NONE : intOption`, and
- one constructor `SOME : int -> intOption`:

  ```
  SOME 8 : intOption
  ```

Datatypes; Constructors carrying values

```
let workload = function
                | SATURDAY -> NONE
                | SUNDAY   -> NONE
                | FRIDAY   -> SOME 6
                | _        -> SOME 8
```

Type constructors

```
type 'a option = | None
                 | Some of 'a
```

defines a *type operator*:

Type constructors

```
type 'a option = | None
                 | Some of 'a
```

defines a *type operator*: For all type t,

t `option`

is a type.

Type constructors

```
type 'a option = | None
                 | Some of 'a
```

defines a *type operator*: For all type t,

t `option`

is a type. Its values are either

- `None`, or
- `Some` v, where v is of type t.

Type constructors

```
type 'a option = | None
                 | Some of 'a
```

defines a *type operator*: For all type t,

t `option`

"option" is not itself a type

is a type. Its values are either

- `None`, or
- `Some` v, where v is of type t.

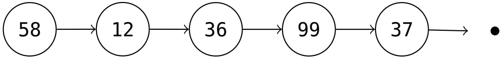
Lists

58	12	36	99	37

A list is

- a collection of many values *of the same type.*

Lists in F#



A list is either

- `[]`, or
- $x :: xs$ where
  - — x is the first element
  - — xs is a list containing the rest

`[1; 2; 3]` is a shorthand for `1 :: 2 :: 3 :: []`

The built-in list type

```
type 'a list = | []
               | (::) of 'a * 'a list
```

The built-in list type

```
type 'a list = | []
               | (::) of 'a * 'a list
```

`list` is a *recursive* datatype: The elements of type α `list` are either

- `[]`, or
- x `::` xs where x is of type α and xs is of type α `list`.

Regular types

We have types for values that contain

- *both* an A *and* a B (tuples),
- *either* an A *or* a B (datatypes), and
- *a sequence of* As (lists).

(Datatypes also give types for trees.)

Recursion

$$2^n = \overbrace{2 \times 2 \times \cdots \times 2}^{n}$$

Recursion

$$2^n = \overbrace{2 \times 2 \times \cdots \times 2}^{n} \times 1$$

$$2^0 = 1$$

Recursion

$$2^n = \overbrace{2 \times 2 \times \cdots \times 2}^{n} \times 1$$

$$2^0 = 1$$

$$2^n = \overbrace{2 \times \underbrace{2 \times \cdots \times 2}_{n-1}}^{n} \times 1$$

Recursion

$$2^n = \overbrace{2 \times 2 \times \cdots \times 2}^{n} \times 1$$

$$2^0 = 1$$

$$2^n = \overbrace{2 \times \underbrace{2 \times \cdots \times 2}_{n-1}}^{n} \times 1$$

$$= 2 \times 2^{n-1}$$

Recursion

$$2^n = \overbrace{2 \times 2 \times \cdots \times 2}^{n} \times 1$$

$$2^0 = 1$$

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 * pow2 (n - 1)
```

$$2^n = \overbrace{2 \times \underbrace{2 \times \cdots \times 2}_{n-1}}^{n} \times 1$$

$$= 2 \times 2^{n-1}$$

Pattern matching

```
let rec pow2 n = match n with
                    | 0 -> 1
                    | m -> 2 * pow2 (m - 1)

// or

let rec pow2 = function
                | 0 -> 1
                | n -> 2 * pow2 (n - 1)
```

Functions on lists

We (almost) always define functions on lists by pattern matching against

- $[\,]$, and
- $x :: xs$

Such functions call themselves recursively on xs.

Functions on lists

We (almost) always define functions on lists by pattern matching against

- `[]`, and
- x `::` xs

Such functions call themselves recursively on xs.

```
let rec f = function
  | []      -> ...
  | x :: xs -> ... f xs ...
```

Computing the length of lists

```
let rec len = function
  | []      ->
```

Computing the length of lists

```
let rec len = function
  | []      -> 0
  | x :: xs ->     len xs
```

Computing the length of lists

```
let rec len = function
  | []      -> 0
  | x :: xs -> 1 + len xs
```

The sum of the elements of a lists

```
let rec sum = function
  | []      ->
```

The sum of the elements of a lists

```
let rec sum = function
  | []       -> 0
  | x :: xs ->     sum xs
```

The sum of the elements of a lists

```
let rec sum = function
  | []      -> 0
  | x :: xs -> x + sum xs
```

Appending two lists

```
let rec append xs ys =
  match xs with
    | []      -> ys
    | x :: xs ->      append xs ys
```

Appending two lists

```
let rec append xs ys =
  match xs with
    | []      -> ys
    | x :: xs -> x :: append xs ys
```

Lists *vs* Arrays

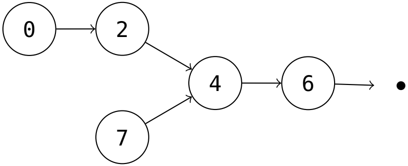
- Lists are *immutable:* Their elements cannot be modified.
  (Instead of modifying a list, we construct a new one!)

- Lists are constructed right-to-left.
  (Given a list *xs* and an element *x*, `x :: xs` is a new list with head *x* and tail *xs*.)

- Two lists may share the same tail.
  (It is important that we cannot modify elements!)

Lists sharing a common tail

```
let L = [4; 6]      // or 4 :: 6 :: []
let x = 0 :: 2 :: L
let y = 7 :: L
```



A function that generates a lists

Function `upto (m, n)` returns `[m; m+1; ...; n]`.

```
let rec upto (m, n) =
  if
```

A function that generates a lists

Function `upto (m, n)` returns `[m; m+1; ...; n]`.

```
let rec upto (m, n) =
  if m > n then
```

A function that generates a lists

Function `upto (m, n)` returns `[m; m+1; ...; n]`.

```
let rec upto (m, n) =
  if m > n then
    []
  else
```

A function that generates a lists

Function `upto (m, n)` returns `[m; m+1; ...; n]`.

```
let rec upto (m, n) =
  if m > n then
    []
  else
    m ::
```

A function that generates a lists

Function `upto (m, n)` returns `[m; m+1; ...; n]`.

```
let rec upto (m, n) =
  if m > n then
    []
  else
    m :: upto (m + 1, n)
```

Quicksort

```
[5; 7; 1; 9; 8; 9; 3; 5; 4; 2]
```

Quicksort

`[5; 7; 1; 9; 8; 9; 3; 5; 4; 2]`

$\downarrow$ Pivot = 5

`[7; 1; 9; 8; 9; 3; 5; 4; 2]`

`partition 5`

`[1; 3; 5; 4; 2]`

`[7; 9; 8; 9]`

Quicksort

[5; 7; 1; 9; 8; 9; 3; 5; 4; 2]

$\downarrow$ Pivot = 5

[7; 1; 9; 8; 9; 3; 5; 4; 2]

partition 5

[1; 3; 5; 4; 2]

$\downarrow$ sort

[1; 2; 3; 4; 5]

[7; 9; 8; 9]

$\downarrow$ sort

[7; 8; 9; 9]

Quicksort

[5; 7; 1; 9; 8; 9; 3; 5; 4; 2]

Pivot = 5

[7; 1; 9; 8; 9; 3; 5; 4; 2]

partition 5

[1; 3; 5; 4; 2]

sort

[1; 2; 3; 4; 5]

[7; 9; 8; 9]

sort

[7; 8; 9; 9]

[1; 2; 3; 4; 5] @ [5] @ [7; 8; 9; 9]

Append everything

[1; 2; 3; 4; 5; 5; 7; 8; 9; 9]

Quicksort

To sort a list

1. pick an element p in the list,
2. partition the remaining elements into
   — a left part containing elements $\leq p$, and
   — a right part containing elements $> p$; then
3. sort these two parts, and
4. return these sorted parts with the p in between.

Partitioning

```
let rec partition p = function
  | []      -> ([], [])
  | y :: ys -> let (l, r) = partition p ys
               if y <= p then
                 (y :: l, r)
               else
                 (l,      y :: r)
```

Quicksort in F#

```
let rec sort = function
  | []          -> []
  | [x]         -> [x]
  | p :: xs     -> let (l,r) = partition p xs
                   sort l @ [p] @ sort r
```

Evaluation as substitute & reduce

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 * pow2 (n - 1)
```

```
pow2 4
```

Evaluation as substitute & reduce

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 ∗ pow2 (n - 1)
```

```
pow2 4
```

$\longmapsto$ (substitute `n = 4`)

```
if   4 = 0
then 1
else 2 ∗ pow2 (4 - 1)
```

Evaluation as substitute & reduce

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 * pow2 (n - 1)
```

```
pow2 4
```

$\longmapsto$ (substitute `n = 4`)

```
if   4 = 0
then 1
else 2 * pow2 (4 - 1)
```

$\longmapsto$ (reduce `4 = 0` to `false`)

```
if   false
then 1
else 2 * pow2 (4 - 1)
```

Evaluation as substitute & reduce

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 * pow2 (n - 1)
```

```
pow2 4
```

$\longmapsto$ (substitute `n = 4`)

```
if   4 = 0
then 1
else 2 * pow2 (4 - 1)
```

$\longmapsto$ (reduce `4 = 0` to `false`)

```
if   false
then 1
else 2 * pow2 (4 - 1)
```

$\longmapsto$ (reduce `if false`)

```
2 * pow2 (4 - 1)
```

Evaluation as substitute & reduce

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 * pow2 (n - 1)
```

```
pow2 4
```

$\longmapsto$ (substitute `n = 4`)

```
if   4 = 0
then 1
else 2 * pow2 (4 - 1)
```

$\longmapsto$ (reduce `4 = 0` to `false`)

```
if   false
then 1
else 2 * pow2 (4 - 1)
```

$\longmapsto$ (reduce `if false`)

```
2 * pow2 (4 - 1)
```

$\longmapsto$ (reduce `4 - 1` to 3)

Evaluation as substitute & reduce

```
let rec pow2 n = if   n = 0
                 then 1
                 else 2 * pow2 (n - 1)
```

```
pow2 4
    ⟼ (substitute n = 4)
if   4 = 0
then 1
else 2 * pow2 (4 - 1)
    ⟼ (reduce 4 = 0 to false)
if   false
then 1
else 2 * pow2 (4 - 1)
    ⟼ (reduce if false)
2 * pow2 (4 - 1)
    ⟼ (reduce 4 - 1 to 3)
```

```
pow2 4  ⟼  2 * pow2 (4 - 1)
        ⟼  2 * pow2 3
        ⟼  2 * (2 * pow2 (3 - 1))
        ⟼  2 * (2 * pow2 2)
        ⟼  2 * (2 * (2 * pow2 (2 - 1)))
        ⟼  2 * (2 * (2 * pow2 1))
        ⟼  2 * (2 * (2 * (2 * pow2 (1 - 1))))
        ⟼  2 * (2 * (2 * (2 * pow2 0)))
        ⟼  2 * (2 * (2 * (2 * 1)))
        ⟼  2 * (2 * (2 * 2))
        ⟼  2 * (2 * 4)
        ⟼  2 * 8
        ⟼  16
```

Functions, F# vs {C, C++, C#, Java, ...}

Function in F#

```
let rec pow2 n =
  if n = 0 then
    1
  else
    2 * pow2(n - 1)
```

Function in imperative languages

```
int pow2(int n) {
  if (n == 0)
    return 1;
  else
    return 2 * pow2(n - 1);
}
```

Iteration vs recursion

Iteration in imperative languages

```
int pow2(int n) {
  int r = 1;
  while (n > 0) {
    r = 2 * r;
    n = n - 1;
  }
  return r;
}
```

Iteration vs recursion

Iteration in imperative languages

```
int pow2(int n) {
  int r = 1;
  while (n > 0) {
    r = 2 * r;
    n = n - 1;
  }
  return r;
}
```

Iteration in F#

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n = loop n 1
```

Iteration vs recursion

Loop becomes recursive function

Iteration in imperative languages

```
int pow2(int n) {
  int r = 1;
  while (n > 0) {
    r = 2 * r;
    n = n - 1;
  }
  return r;
}
```

Iteration in F#

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n = loop n 1
```

Iteration vs recursion

Iteration in imperative languages

```
int pow2(int n) {
  int r = 1;
  while (n > 0) {
    r = 2 * r;
    n = n - 1;
  }
  return r;
}
```

Iteration in F#

Loop modifies n and r

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n = loop n 1
```

Iteration vs recursion

Iteration in imperative languages

```
int pow2(int n) {
  int r = 1;
  while (n > 0) {
    r = 2 * r;
    n = n - 1;
  }
  return r;
}
```

Iteration in F#

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n = loop n 1
```

When done, return r

Iteration vs recursion

Iteration in imperative languages

```
int pow2(int n) {
  int r = 1;
  while (n > 0) {
    r = 2 * r;
    n = n - 1;
  }
  return r;
}
```

Iteration in F#

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n = loop n 1
```

Start here

Evaluating the iterative `pow2i`

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n =
  loop n 1
```

```
pow2i 4  ⟼  loop 4 1
         ⟼  loop (4 - 1) (2 * 1)
         ⟼  loop 3 2
         ⟼  loop (3 - 1) (2 * 2)
         ⟼  loop 2 4
         ⟼  loop (2 - 1) (2 * 4)
         ⟼  loop 1 8
         ⟼  loop (1 - 1) (2 * 8)
         ⟼  loop 0 16
         ⟼  16
```

Evaluating the iterative pow2i

```
let rec loop n r =
  if n > 0 then
    loop (n - 1) (2 * r)
  else
    r
let pow2i n =
  loop n 1
```

$$
\begin{aligned}
\underline{\texttt{pow2i 4}} &\longmapsto \underline{\texttt{loop 4 1}} \\
&\longmapsto \texttt{loop (}\underline{\texttt{4 - 1}}\texttt{) (}\underline{\texttt{2 * 1}}\texttt{)} \\
&\longmapsto \underline{\texttt{loop 3 2}} \\
&\longmapsto \texttt{loop (}\underline{\texttt{3 - 1}}\texttt{) (}\underline{\texttt{2 * 2}}\texttt{)} \\
&\longmapsto \underline{\texttt{loop 2 4}} \\
&\longmapsto \texttt{loop (}\underline{\texttt{2 - 1}}\texttt{) (}\underline{\texttt{2 * 4}}\texttt{)} \\
&\longmapsto \underline{\texttt{loop 1 8}} \\
&\longmapsto \texttt{loop (}\underline{\texttt{1 - 1}}\texttt{) (}\underline{\texttt{2 * 8}}\texttt{)} \\
&\longmapsto \underline{\texttt{loop 0 16}} \\
&\longmapsto \texttt{16}
\end{aligned}
$$

Notice: Only two memory cells required