# INF231: Functional Algorithmic and Programming

## Lecture 1: Introduction, simple expressions and simple types

Academic Year 2019 - 2020

## The right vision about computer science

Computer science is NOT about:

- using a computer
- fix a computer
- using software or internet (Facebook, Google, Word, . . . )

Among other things, computer science is about:

- understanding computers
- understanding computation
- designing (efficient) methods to compute

*“Computer science is no more about computers than astronomy is about telescopes.”*

Edsger Wybe Dijkstra

## About algorithms and algorithmic
A central and basic concept in computer science

Algorithmic, the science of algorithms, consists in:

- Automating methods purposed to solve a problem
- Study correctness, completeness, and efficiency of a solution

Algorithm: step-by-step instructions for a calculation

Four styles (among others) can be used to express algorithms:

- **imperative-style**: a list of actions (known)
- object-oriented: objects and their interactions are first-class citizens
- logical languages: predicates are first-class citizens
- functional-style: closer to mathematical concepts

Then we turn algorithms into programs using a programming language

## Imperative vs functional algorithmic styles

On examples

### Example (GCD, Greatest Common Divisor, of two integers *a* and *b*)

Can be computed using the remainder of the euclidian division of *a* by *b*

Imperative style (C)

```c
int gcd (int a, int b) {
   int r;
   while ((r=a%b)!=0) {
      a = b;
      b = r;
   }
   return b;
}
```

Functional style (OCaml)

```ocaml
let rec gcd (a:int) (b:int):int
   = let r = a mod b in
      if r = 0 then b
      else gcd b r
```

- the gcd of 8 and 12 is 4
- code is shorter
- nothing is modified
- closer to the mathematical procedure

- **Write this fuction in Python ?**

# Imperative vs functional algorithmic styles

On examples

## Example (Factorial of an integer)

**Imperative style (C)**

```
int fact (int n) {
   int res;
   if (n==0) {return 1;}
   else {
      res = 1;
      for (i=1;i<=n;i++) {
         res = res *i;
      }
      return res;
}
```

**Functional style (OCaml)**

```
let rec fact (n:int):int =
   if (n=0 || n=1) then 1
   else n * fact (n-1)
```

$$0! = 1$$

$$n! = n * (n - 1)!$$

- fact(4) = 4 x 3 x 2 x 1 = 24 ; 4! = 4 x 3!
- code is shorter
- exactly the mathematical definition
- easier to understand

- **Write this fuction in Python ?**

## Imperative vs functional algorithmic styles
### The killing example

**Example (Yielding affine functions)**

Given two integers *a* and *b*, compute/return the function $x \mapsto a * x + b$

| Imperative style (C) | Functional style (OCaml) |
|---|---|
| a nightmare... (3 pages of code) | `let affine (a:int) (b:int):int -> int = fun x -> a*x+b` |

```
let affine (a:int) (b:int):int -> int
   = fun x -> a*x+b
```

## Review

### Example (calculate sum of n numbers using array)

**Imperative style (C)**

```
int n, sum = 0, i,
       myArr[50];

scanf("%d", &n);
for (i = 0; i < n; i++)
 {
    scanf("%d", &myArr[i]);
    sum = sum + myArr[i];
 }
```

**Recursion style (C)**

```
int calSum(int myArr[],
                    int n) {
   static int sum = 0;

   if (n == 0)
      return sum;

   sum = sum + myArr[n-1];

   return calSum(myArr, --n);
}
```

- **Write this fuction in Python ?**

## Le language [O]Caml

- O Caml: a general-purpose programming language, safety, reliability
- O Caml: easy to learn, use. Caml supports functional, imperative, and object-oriented programming styles.
- `http://caml.inria.fr/ocaml`

## Le language [O]Caml and Functional languages in general

in a nutshell

Result of the fruitful collaboration of mathematicians and computer scientists:

- they have the rigor of mathematics
- they rely on few but powerful concepts ($\lambda$-calculus)
- they are as expressive as other languages (Turing complete)
- they favor efficient, consise and effective algorithms
- they insist on typing

Example (OCaml in nature)

Unison
File Synchronizer

LexiFi

...

## About OCaml and functional languages in general

### Features and Advantages

Features:

| | |
|---|---|
| **Fun**ctional: | ▶ functions are first-class values and citizens<br>▶ highly flexible with the use of functions: nesting, passed as argument, storing |
| strongly typed: | ▶ everything is typed at compile time<br>▶ syntactic constraints on programs |
| type inference: | “types automatically computed from the context” |
| polymorphic: | “generic functions” |
| pattern-matching: | “a super `if`” |

Advantages:

| | |
|---|---|
| Rigorous: | closer to mathematical concepts |
| More concise: | less mistakes |
| Typing is a central concept: | better type-safe than sorry |

## Primitive types and basic expressions

### `int`: the integers

The set of signed integers $\mathbb{Z}$, e.g., $-10, 2, 0, 3, 9 \ldots$

Several alternate forms:

| | |
|---|---|
| ddd ... | an int literal specified in decimal |
| *0o*ooo ... | an int literal specified in octal |
| *0b*bbb... | an int literal specified in binary |
| *0x*hhh ... | an int literal specified in hexadecimal |

where *d* (resp. *o*, *b*, *h*) denotes a decimal (resp. octal, binary, hexadecimal) digit

Usual operations:

| | | | |
|---|---|---|---|
| `-i` | negation | `lnot` | bit-wise inverse |
| `i + j` | addition | `i lsl j` | logical shift left |
| `i - j` | substraction | `i lsr j` | logical-shift right |
| `i * j` | multiplication | `i land j` | bitwise-and |
| `i / j` | division | `i lor j` | bitwise-or |
| `i mod j` | remainder | `i lxor j` | bitwise exclusive-or |

DEMO: integers

## Primitive types and basic expressions

### `float`: the real numbers

The set of real numbers $\mathbb{R}$ (an approximation actually): dynamically scaled floating point numbers
Requires at least either:

- a decimal point, or
- an exponent (base 10), prefixed by an *e* or *E*

**Remark** Not exact computation □

### Example

0.2, 2`e7`, 1`E10`, 10.3`E2`, 33.23234`E`(−1.5), 2.

Usual operators:

| | |
|---|---|
| `-.x` | floating-point negation |
| `x +. y` | floating-point addition |
| `x -. y` | floating-point subtraction |
| `x *. y` | float-point multiplication |
| `x /. y` | floating-point division |
| `int_of_float x` | float to int conversion |
| `float_of_int x` | int to float conversion |

DEMO: float

## Primitive types and basic expressions

### `bool`: the Booleans

The set of truth-values $\mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}$

Some operators on Booleans:

| | |
|---|---|
| `not` | logical negation |
| && | logical conjunction (short-circuit) |
| \|\| | logical disjunction (short-circuit) |

DEMO: operators using Booleans

## Primitive types and basic expressions

### `bool`: the Booleans

Some operations returning a Boolean

| | |
|---|---|
| `x = y` | x is *equal* to y |
| `x == y` | x is *identical* to y |
| `x != y` | x is not identical to y |
| `x <> y` | x is not equal to y |
| `x < y` | x is less than y |
| `x <= y` | x is not greater than y |
| `x >= y` | x is not lesser than y |
| `x > y` | x is greater than y |

DEMO: operators returning Booleans

**Remark** Distinction between == and =:

- = is *structural* equality (compare the structure of arguments)
- == is *physical* equality (check whether the arguments occupy the same memory location)
- Returns the same results on basic types: int, bool, char

Hence `e1 == e2` implies `e1 = e2` □

DEMO: illustration of the difference between = and ==

## Primitive types and basic expressions

### `char`: the Characters

The set of characters $Char \subseteq \{'a', 'b', \ldots, 'z', 'A', \ldots, 'Z'\}$
Contains also several escape sequences:

| | |
|---|---|
| `'\\'` | backslash character itself |
| `'\''` | single-quote character |
| `'\t'` | tabulation character |
| `'\r'` | carriage return character |
| `'\n'` | new-line character |
| `'\b'` | backspace character |

Conversion from int to char (and vice-versa): a char can be represented using its ASCII code:

- `Char.code`: returns the ASCII code of a character
- `Char.chr`: returns the character with the given ASCII code

From lower to upper-case and vice-versa:

- `Char.lowercase`
- `Char.uppercase`

DEMO: char

## Primitive types and basic expressions

### `unit`: the singleton type

Simplest type that contains one element ()

Used by side-effect functions (every function should return a value)

**Remark** Similar to type `void` in C □

**Rarely used!**

DEMO: type unit

## More on operators

### Operators have a type

Constraining the arguments and results:

- order
- number

$\hookrightarrow$ the "signature of the operator"

Operators are functions, i.e., values (hence they have a type).

Consider an operator `op`:

$$\begin{array}{ll} \texttt{arg1} & \textit{type}_1 \\ \texttt{arg2} & \textit{type}_2 \\ \ldots & \ldots \\ \texttt{argn} & \textit{type}_n \\ \texttt{result} & \textit{type}_r \end{array} \quad \Rightarrow \quad \underbrace{\text{type}_1 \to \text{type}_2 \to \ldots \to \text{type}_n \to \textit{type}_r}_{= \text{ type of } \texttt{op}}$$

Example (Types of some operators)

$+$ : `int` $\to$ `int` $\to$ `int`

$=$ : `int` $\to$ `int` $\to$ `bool`

$<$ : `int` $\to$ `int` $\to$ `bool`

...

DEMO: type of operators

## More on operators

### precedences and associativity

Remainder about associativity:

- right associativity: `a op b op c` means `a op ( b op c)`
- left associativity: `a op b op c` means (`a op b`) `op c`

Precedences of operators on the basic types, in **increasing order**:

| Operators | | | | | | | | Associativity |
|---|---|---|---|---|---|---|---|---|
| \|\| | && | | | | | | | left |
| = | == | != | <> | < | <= | > | >= | left |
| + | - | +. | -. | | | | | left |
| * | / | *. | /. | mod | land | lor | lxor | left |
| `lsl` | `lsr` | `asr` | | | | | | left |
| `lnot` | | | | | | | | left |
| - | -. | | | | | | | right |

## More on Typing
### About OCaml type system

Typing is a mechanism/concept aiming at:

- avoiding errors
- favoring *abstraction*
- checking that expressions are sensible, e.g.
  - 1 + `yes`
  - `true` $*$ 42

Type checking in OCaml: OCaml is strictly and statically typed

- strict: no implicit conversion between types nor type coercion (force)
- static: checking performed before execution

Type inference: for any expression *e*, OCaml (automatically and systematically) computes the type of *e*:

### Example (Type system on integers and floats)

- Two sets of distinct operations:
  - integers (+,$-$,$*$)
  - floats (+.,$-$., $*$.)
- No implicit conversion between them, e.g., 1+ 0.42 yields an error

## More on Typing
### About OCaml type system (ctd)

OCaml is a safe programming language:

- Programs never go wrong at runtime
- Easier to write correct programs: many errors are detected

**Remark** Comparison with C:

- C is *weakly typed*: values can be coerced
- a lot of runtime errors, e.g., segmentation-fault, bus-error, etc. . .

□

*"Better type-safe than sorry"*

## The language constructs

if . . . then . . . else . . .

An expression defined using an alternative (or a conditional) control structure

```
if cond then expr1 else expr2
```

- the result is a value
- `cond` should be a Boolean expression
- `expr1` and `expr2` should be of the same type

**Remark** The else branch cannot be omitted unless the whole `expr1` is of type unit (hence the whole expression is of type unit) □

DEMO: if. . . then. . . else. . .

## Running your code
### Compilation vs Interpretation

Two ways to interact/evaluate/execute your code: compilation and interactive interpretation

Compiling:

- Place your program in a `.ml` file
- Use one of the compilers:
  - `ocamlc`: compiles to byte-code
  - `ocamlopt`: compiles to native machine code

Interpretation:

- Type `ocaml`
- Directly type your expression

**Remark**

- Byte-code is compiled faster but runs slower
- Native machine code is compiled slower but runs faster

DEMO: compiling vs interpreting, compiler options

## Summary and Assignment

### Summary

- Basic types and operations:

| type | operations | constants |
|---|---|---|
| Booleans | `not, &&, \|\|` | `true, false` |
| integers | `+,-,*,/,mod` | `..., -1, 0, 1, ...` |
| floats | `+.,-.,*.,/.` | `0.4, 12.3, 16. , 64.` |

- `if`...`then`...`else` constcuct
- OCaml type system
- Compilation / Interpretation

### Assignment 1

- Write a program to sum of two integers ? two float ?
- Write a program to check an integer number is positive or negative (use *if…then...else*) ?
- Write a program to compare two integers (equal, greater, less) ?