

INF231:
Functional Algorithmic and Programming
Lecture 2: Identifiers and functions

Academic Year 2020 - 2021

$f(x)$



Identifiers

The notion of identifier

A fundamental concept of programming languages: associating a value to a name (**an identifier**)

Remark “Close” to the notion of *variable* but has fundamental differences! □

Some rules when defining identifiers:

- ▶ Maximal length: 256 characters
- ▶ Must begin with a non-capital letter
- ▶ No blanks
- ▶ Case-sensitive
- ▶ Should have a meaningful name

Example (Identifiers (valid and invalid))

- | | |
|-------------------|----------------|
| ▶ speed ✓ | ▶ s ✓ X |
| ▶ Speed X | ▶ 3m X |
| ▶ average speed X | ▶ temporary3 ✓ |
| ▶ average_speed ✓ | |

Identifiers: Global definition

Syntax of a *global definition*

```
let identifier = expression
```

↔ the value of `expression` is bound/linked to `identifier`

Type of the identifier is the type of the evaluated expression

Definition is **global**: it can be used

- ▶ in other definitions
- ▶ in the rest of the program

Simultaneous definitions:

```
let ident1 = expr1  
and ident2 = expr2  
...  
and identn = exprn
```

Example

- ▶ `let x = 1`
- ▶ `let i = 1`
- ▶ `let y = 2`
- ▶ `let i = i+1`

DEMO: global definitions

Identifiers: Local definition

Example (Motivating example)

How to compute $e = (2 \cdot 3 \cdot 4) \cdot (2 \cdot 3 \cdot 4) + (2 \cdot 3 \cdot 4) + 2$?

\hookrightarrow prod = $(2 \cdot 3 \cdot 4)$

$\hookrightarrow e = \text{prod} \cdot \text{prod} + \text{prod} + 2$

\hookrightarrow prod is *local* to e

Syntax of a *local* definition:

```
let identifier = expression1 in expression2
```

\hookrightarrow the value of expression1 is **permanently** bound/linked to identifier
(only) when evaluating expression2

Can be nested:

```
let id1=expr1 in
  let id2=expr2 in
    ...
    let idn = exprn ... in expr
```

Works with simultaneous definitions:

```
let id1=expr1
  and id2=expr2
  ...
  and idn = exprn ... in expr
```

DEMO: local definitions

Functions

Introduction

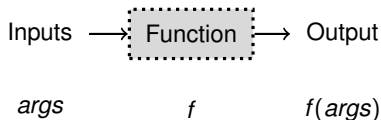
So far, we have considered:

- ▶ expressions
- ▶ pre-defined operators

Defining our own functions: a piece of code with a specific job

Motivations:

- ▶ code readability
- ▶ its job can be more elaborated than the job of pre-defined functions
- ▶ being able to execute this code from several locations



Functions in functional languages

- ▶ No side-effect (contrarily to C)
- ▶ Close to mathematical functions
- ▶ First-class objects: they are values \Rightarrow they have a *type*

Functions: functions with one argument

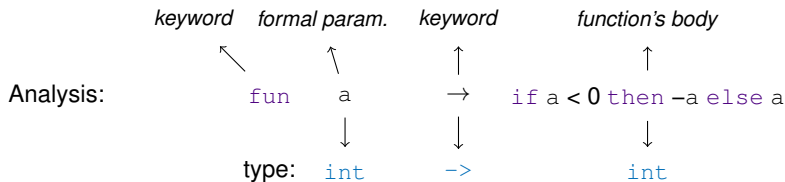
On an example

Example (Absolute value from a mathematical/abstract point of view)

$$\begin{aligned} \mathbb{Z} &\rightarrow \mathbb{N} \\ a &\mapsto \text{if } a < 0 \text{ then } -a \text{ else } a \end{aligned}$$

Example (Absolute value in OCaml)

```
fun a → if a < 0 then -a else a  
or function a → if a < 0 then -a else a  
or fun/function (a:int) → if a < 0 then -a else a
```



Remark This function is **anonymous**, i.e., it does not have a name □

DEMO: anonymous functions

Functions

How to define them

Naming a function allows to reuse it

Example (Defining the function absolute value)

```
let abs = fun (a:int) → if a < 0 then -a else a  
or let abs a = if a < 0 then -a else a  
or let abs (a:int) = if a < 0 then -a else a  
or let abs (a:int):int = if a < 0 then -a else a
```

DEMO: defining functions

Exercise

Define the function `square: int → int`

Functions

How to use them

As in mathematics, the result of applying f to x is $f(x)$

Example

- ▶ `abs(2)`
- ▶ `abs(2 - 3)`
- ▶ `abs 2` (parenthesis can be omitted)

Application of a function

`expr1 expr2`

Typing: **if** `expr1` has type `t1->t2` and `expr2` has type `t1` } **then** `expr1 expr2` has type `t2`

Functions: Generalization to functions with several arguments

Example (Surface area of a rectangle)

- ▶ Needs 2 parameters: length and width (floats)

- ▶ definition:

```
let surface (x:float) (y:float):float = x *. y
let surface (length:float) (width:float):float = length *.
width
```

- ▶ usage: `surface 2.3 1.2`

Definition of a Function with n parameters

```
let fct_name (p1:t1) (p2:t2) ... (pn:tn) : t = expr
```

- ▶ p_1, \dots, p_n are *formal* parameters
- ▶ Type of `fct_name` is `t1 -> t2 -> ... -> tn -> t`

Using a Function with n parameters

```
fct_name e1 e2 ... en
```

- ▶ e_1, \dots, e_n are *effective* parameters
- ▶ Type of `fct_name e1 e2 ... en` is `t`
if t_i is the type of e_i and `fct_name` is of type `t1 -> t2 -> ... -> tn -> t`

Functions: SPECIFICATION and IMPLEMENTATION

In this module (and in your future), it is very important to distinguish two concepts/stages about defining functions (and programs in general)

Specification:

A description of **what** it is expected to do/ the job

- ▶ at an abstract level
- ▶ should be precise
- ▶ close to maths description in fun programming
- ▶ illustrate the function with some *interesting* examples

Implementation:

The description of **how** it is done

- ▶ the OCaml code

Defining a function: Specification **AND THEN** Implementation

Has many advantages (how big software is developed):

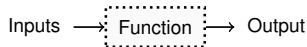
- ▶ re-usability
- ▶ you will save a lot of time
- ▶ thinking before acting
- ▶ you will have a better grade

A contract:



Consists of:

- ▶ description
- ▶ signature
- ▶ examples



Defining functions: some examples

Example (Defining the function absolute value)

- ▶ Specification:
 - ▶ The function absolute value *abs* takes an integer n as a parameter and returns n if this integer is positive or $-n$ if this integer is negative. The function absolute value always returns a positive integer.
 - ▶ Profile: $\mathbb{Z} \rightarrow \mathbb{N}$
 - ▶ Example: $abs(1) = 1$, $abs(0) = 0$, $abs(-2) = 2$
- ▶ Implementation: `let abs (a:int) = if a < 0 then -a else a`

Example (Defining the function square)

- ▶ Specification:
 - ▶ The function square *sq* takes an integer n as a parameter and returns $n * n$.
 - ▶ Profile: $\mathbb{Z} \rightarrow \mathbb{N}$
 - ▶ Example: $sq(1) = 1$, $sq(0) = 0$, $sq(3) = 9$, $sq(-4) = 16$
- ▶ Implementation: `let sq (n:int) = n*n`

Some exercises

A piece of algorithmic

Exercise

Define the function `my_max` returning the maximum of two integers

Exercise

Define functions:

- ▶ `square: int → int`
- ▶ `sum_square: int → int → int`

s.t. `sum_square` computes the sum of the squares of two numbers

Problem: Olympic mean

Computing the mean of 4 grades (or values), by suppressing the highest and lowest one

1. Propose a type for the function `mean`
2. Propose an algorithm, by supposing that you have two functions `min4` and `max4`, which compute respectively the minimum and the maximum of 4 integers
3. Define functions `min4` et `max4`, using `min` and `max`